Dynamic Slicing: a Generic Analysis Based on a Natural Semantics Format

Valérie Gouranton, Daniel Le Métayer

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Dynamic slicing: a generic analysis based on a natural semantics format

Valérie Gouranton et Daniel Le Métayer

N° 3375

Mars 1998

THÈME 2

ISSN 0249-6399

INRIA
RENNES

Dynamic slicing: a generic analysis based on a natural semantics format

Valérie Gouranton et Daniel Le Métayer

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lande

Rapport de recherche n3375 — Mars 1998 — 40 pages

Abstract: Slicing analyses have been proposed for different programming languages. Rather than defining a new analysis from scratch for each programming language, we would like to specify such an analysis once for all, in a language-independent way, and then specialise it for different programming languages. In order to achieve this goal, we propose a notion of natural semantics format and a dynamic slicing analysis format. The natural semantics format formalises a class of natural semantics and the analysis format is a generic, language-independent, slicing analysis. The correctness of the generic analysis is established as a relation between the derivation trees of the original program and the slice. This generic analysis is then instantiated to several programming languages conforming the semantics format (an imperative language, a logic programming language and a functional language), yielding a dynamic slicing analyser for each of these languages.

Key-words: dynamic slicing analysis, natural semantics, proof tree, correctness, systematic derivation.

(Résumé : tsvp)

Unit´e de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
T´el´ephone : 02 99 84 71 00 - International : +33 2 99 84 71 00
T´el´ecopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

L'élagage dynamique : une analyse générique basée sur un format de sémantique naturelle

Résumé : Il existe un certain nombre d'analyses d'élagage pour différents langages de programmation. Au lieu de définir une nouvelle analyse pour chacun des langages, nous aimerions spécifier l'analyse d'élagage indépendamment du langage considéré, puis l'instancier à plusieurs langages. Pour atteindre cet objectif, nous proposons un format de sémantique naturelle qui formalise une classe de sémantiques naturelles et nous définissons un format d'analyse dynamique d'élagage qui est générique au sens où il est indépendant du langage. Nous établissons ensuite la correction de l'analyse générique d'élagage sous forme de relation entre les arbres de dérivation du programme initial et du *slice* (le programme transformé). L'analyse générique d'élagage est alors instanciée à plusieurs langages de programmation (un langage impératif, un langage de programmation logique et un langage fonctionnel) en conformité avec le format de sémantique. Nous obtenons par la suite un analyseur dynamique pour chacun de ces langages.

Mots-clé : analyse dynamique d'élagage, sémantique naturelle, arbre de dérivation, correction, dérivation systématique.

1 Introduction

One motivation for this work stems from the observation that slicing analyses have been proposed for different programming languages. Other examples of analyses of general interest include sharing analyses, data flow analyses ... Some of these analyses are static (typically for compiler optimisation), others can be either static or dynamic (for example in applications to debugging). Depending on their nature (static or dynamic) and the programming language they apply to, these analyses are described in different ways and with a variable concern for formalisation. As a consequence, it is difficult to factorise efforts in the definition of analyses and one is often led to invent new analyses which could in fact be described as variations or adaptations of existing ones. What we would rather like is to be able to specify such general purpose analyses once for all, in a language-independent way, and then specialise them for different programming languages.

Two main problems have to be solved in order to be able to define an analysis in a language-independent way:

1. We must find an appropriate basis for defining the semantics of the programming language. By appropriate, we mean general enough to encompass a wide variety of programming languages and precise enough to support useful analyses.

2. We must provide a way to define generic analyses and state their correctness with respect to the semantics of the programming language.

In order to achieve the first goal, we propose a notion of natural semantics format. As argued in [8], natural semantics [3, 6] are a good starting point for the definition of static analyses because they are both structural (compositional) and intensional. They are structural because the semantics of a phrase in the programming language is derived from the semantics of subphrases; they are intensional because the derivation tree that is associated to a phrase in the programming language contains the intermediate results (the semantics of subphrases). These qualities are significant in the context of program analysis because compositionality leads to more tractable proof techniques and intensionality makes it easier to establish the connection between the result of the analysis and its intended use. Our natural semantics format is the formalisation of a class of natural semantics. This class is general enough to include the major sequential programming styles (imperative, logic and functional).

As far as the second goal is concerned, we believe that the most natural way to specify an analysis is to characterise it with respect to the intended use of its result. Analyses are generally performed to check assumptions about the behaviour of the program at specific points of its execution or to enable program optimisations. In both cases the intention of the analysis can be expressed in terms of a transformation and a relation. The transformation depends on the result of the analysis and the relation establishes a correspondence between the semantics of the original program and the transformed program. For example, in the case of a program analysis for compiler optimisation the transformation expresses the optimisation that is allowed by the information provided by the analysis and the relation is the equality between the final results (or outputs) of the original and the transformed program. Note that it is not always the case that the relation is the equality: a counter-example is slicing analysis whose aim is to build a slice that behaves like the original program only with respect to specific program points and variables.

As mentioned before, some of the analyses that we want to specify are dynamic and others are static. There is no real reason why these two categories of analyses should be seen as belonging to different worlds. In the paper we focus on dynamic analysis, considering that static analysis can be obtained in a second stage as an abstract interpretation of the dynamic analysis. We outline this derivation in the conclusion.

In the next section, we present our natural semantics format and illustrate it with the specification of a small imperative language. Section 3 introduces slicing and proposes a generic slicing analyser together with its intensional correctness property. The correctness proof is detailed in Appendix

1. We show how the generic analyser of section 3 can be instantiated to derive a slicing analyser for an imperative language (section 4), a logic programming language (section 5) and a functional programming language (section 6). Related work and avenues for further research are discussed in section 7 and section 8 respectively.

2 The natural semantics format

The natural semantics of a language is a set of axioms and inference rules that define a relation between a context, a term in the programming language and a result. A natural semantics derivation tree has the form:

$$\text{Proof-Tree} = \quad [\text{RN}] \qquad \frac{\text{Proof-Tree}_1 \quad ... \quad \text{Proof-Tree}_n}{\text{STT}}$$

where RN is the name of the rule used to derive the conclusion STT and STT is a statement, that is to say a triple consisting of a context, a term and a result.

In order to make formal manipulations easier, we express the construction of natural semantics derivation trees in a functional framework. Let `C` be the set of contexts, `T` the type of terms of the language and `PT` the type of derivation trees. The semantic function S is a partial function of type:

$$\texttt{C} \times \texttt{T} \rightarrow \texttt{PT}$$

with

$$\begin{array}{lcl} \texttt{PT} & = & \texttt{STT} \times (\mathit{list}\ \texttt{PT}) \times \texttt{RN} \\ \texttt{STT} & = & \texttt{C} \times \texttt{T} \times \texttt{NF} \\ \texttt{T} & = & \texttt{PP} \times \texttt{I} \end{array}$$

Derivation trees are made of a statement (the conclusion), a list of derivation trees (the premises) and the name of rule applied to derive the conclusion. We assume that a term is a pair of a program point and an expression. `STT` denotes the type of statements, `RN` rule names, `NF` normal forms (program results), `PP` program points and `I` expressions.

The important issue about the type of the semantic function is that it returns the whole natural semantics derivation tree, rather than just the result of the program. This choice makes it easier to define intensional analyses. The fact that we describe the semantics in a functional framework does not prevent us from dealing with non deterministic languages, as we show in section 5. This is because we can use `NF` and `C` to represent sets of possible results and contexts.

We use the notation X.`ty` to denote the field of type `TY` of X. For example, we will make intensive use of the following expressions in the rest of the paper:

	type	meaning
PT.`stt`	`STT`	conclusion of PT
PT.`lpt`	(*list* `PT`)	premisses of PT
PT.`rn`	`RN`	name of the rule used at the root of PT
PT.`stt.c`	`C`	context of the conclusion sequent of PT
PT.`stt.t.i`	`I`	term of the conclusion sequent of PT
PT.`stt.t.pp`	`PP`	program point of the conclusion sequent of PT
PT.`stt.nf`	`NF`	normal form of the conclusion sequent of PT

In order to go further and be able to define generic program analyses based on a natural semantics, we need to know more about the form of a natural semantics function $\mathcal{S}$. To this end, we define in Figure 1 a natural semantics format that will serve as the basis for the developments of this paper.

To make this definition more concrete, we show in Figure 3 the natural semantics of a small imperative language using the usual inference rule presentation and we present in Figure 4 the natural semantics in our format. The syntax of the language is defined in Figure 2. To make it easier to see why the semantics does indeed conform to our format, we show in Figure 5 the instantiations of

$$
\begin{array}{l}
\mathcal{S}\,(C,T) = \texttt{case } T \texttt{ of} \\
\qquad [(\pi, op^K(E,\overline{U}))\ \texttt{and}\ \mathcal{P}^k(C,E)\ : \\
\qquad\quad \texttt{let}\quad C_i \;=\; \mathcal{C}_i^k(C,\overline{R},E) \\
\qquad\qquad\quad\ \ T_i \;=\; \mathcal{T}_i^k(C,\overline{R},T) \\
\qquad\qquad\quad\ \ PT_i \;=\; \mathcal{S}\,(C_i,T_i) \\
\qquad\qquad\quad\ \ R_i \;=\; PT_i.\texttt{stt.nf} \\
\qquad\quad \texttt{in}\ ((C,T,\mathcal{F}^k(C,\overline{R},E)),\overline{PT},k)]^*
\end{array}
$$

Figure 1: The natural semantics format

$$
\begin{array}{lcl}
U & := & (PP, I) \\
I & := & U_1; U_2 \mid Id := Exp \mid \texttt{cond}\ (Exp, U_1, U_2) \mid \texttt{while}\ Exp\ \texttt{do}\ U_1 \\
Exp & := & Id \mid B \mid Exp_1\ \underline{\texttt{Op}}\ Exp_2 \\
B & := & \textbf{true} \mid \textbf{false} \mid n
\end{array}
$$

Figure 2: Syntax of a small imperative language

$$
[\mathrm{k}_1]\quad \frac{C \vdash U_1 \to C_1 \quad C_1 \vdash U_2 \to C_2}{C \vdash (\pi, U_1; U_2) \to C_2}
$$

$$
[\mathrm{k}_2]\quad C \vdash (\pi, Id := Exp) \to C[Val(Exp,C)/Id]
$$

$$
[\mathrm{k}_3]\quad \frac{C \vdash U_1 \to C_1}{C \vdash (\pi, \texttt{cond}\ (Exp, U_1, U_2)) \to C_1} \qquad Val(Exp,C) = \textbf{true}
$$

$$
[\mathrm{k}_4]\quad \frac{C \vdash U_2 \to C_2}{C \vdash (\pi, \texttt{cond}\ (Exp, U_1, U_2)) \to C_2} \qquad Val(Exp,C) = \textbf{false}
$$

$$
[\mathrm{k}_5]\quad \frac{C \vdash U_1 \to C_1 \quad C_1 \vdash (\pi, \texttt{while}\ Exp\ \texttt{do}\ U_1) \to C_2}{C \vdash (\pi, \texttt{while}\ Exp\ \texttt{do}\ U_1) \to C_2} \qquad Val(Exp,C) = \textbf{true}
$$

$$
[\mathrm{k}_6]\quad C \vdash (\pi, \texttt{while}\ Exp\ \texttt{do}\ U_1) \to C \qquad Val(Exp,C) = \textbf{false}
$$

Figure 3: Natural semantics of a small imperative language

$\mathcal{S}(C,T) = \texttt{case}\ \ T\ \texttt{of}$

$(\pi, U_1; U_2)$:
$\texttt{let}\quad PT_1 = \mathcal{S}(C,U_1)$
$\qquad\ R_1 = PT_1.\texttt{stt.nf}$
$\qquad\ PT_2 = \mathcal{S}(R_1,U_2)$
$\qquad\ R_2 = PT_2.\texttt{stt.nf}$
$\texttt{in}\quad ((C,T,R_2),[PT_1,PT_2],k_1)$

$(\pi, Id := Exp) : ((C,T,C[Val(Exp,C)/Id]),\texttt{nil},k_2)$

$(\pi, \texttt{cond}\ (Exp,U_1,U_2))\ \texttt{and}\ Val(Exp,C) = \textbf{true}$:
$\texttt{let}\quad PT_1 = \mathcal{S}(C,U_1)$
$\qquad\ R_1 = PT_1.\texttt{stt.nf}$
$\texttt{in}\quad ((C,T,R_1),[PT_1],k_3)$

$(\pi, \texttt{cond}\ (Exp,U_1,U_2))\ \texttt{and}\ Val(Exp,C) = \textbf{false}$:
$\texttt{let}\quad PT_1 = \mathcal{S}(C,U_2)$
$\qquad\ R_1 = PT_1.\texttt{stt.nf}$
$\texttt{in}\quad ((C,T,R_1),[PT_1],k_4)$

$(\pi, \texttt{while}\ Exp\ \texttt{do}\ U_1)\ \texttt{and}\ Val(Exp,C) = \textbf{true}$:
$\texttt{let}\quad PT_1 = \mathcal{S}(C,U_1)$
$\qquad\ R_1 = PT_1.\texttt{stt.nf}$
$\qquad\ PT_2 = \mathcal{S}(R_1,T)$
$\qquad\ R_2 = PT_2.\texttt{stt.nf}$
$\texttt{in}\quad ((C,T,R_2),[PT_1,PT_2],k_5)$

$(\pi, \texttt{while}\ Exp\ \texttt{do}\ U_1)\ \texttt{and}\ Val(Exp,C) = \textbf{false}$:
$\qquad((C,T,C),\texttt{nil},k_6)$

Figure 4: Semantics of the imperative language in the natural semantics format

k	$\mathcal{C}_1^k$	$\mathcal{C}_2^k$	$\mathcal{T}_1^k$	$\mathcal{T}_2^k$	$\mathcal{F}^k$	P^k
k_1	C	R_1	U_1	U_2	R_2	**true**
k_2					$C[Val(Exp, C)/Id]$	**true**
k_3	C		U_1		R_1	$Val(Exp, C) =$ **true**
k_4	C		U_2		R_1	$Val(Exp, C) =$ **false**
k_5	C	R_1	U_1	`while` Exp `do` U_1	R_2	$Val(Exp, C) =$ **true**
k_6					C	$Val(Exp, C) =$ **false**

Figure 5: Instantiation of the natural semantics format for the imperative language

the different functions. We assume a function Val of type $\texttt{C} \times \texttt{Exp} \rightarrow \texttt{NF}$ to evaluate the value of an expression.

The semantics function of Figure 1 takes two arguments (the context C and the term T) and it returns a derivation tree. The derivation tree contains the conclusion $(C, T, \mathcal{F}^k(C, \overline{R}, E))$, the list of subtrees $\overline{PT}$ and the name k of the rule used to derive the conclusion. We use the notation $\overline{X}$ for $(X_1, \ldots, X_n)$. The body of the function is a list of cases selected by pattern matching on the form of the term and the satisfaction of side conditions. The meta-notation $[\ldots]^*$ is used to denote the repetition of the enclosed piece of text (see Figure 4). Each case k corresponds to an inference rule (or an axiom) in the natural semantics. Because of side conditions, each operator op^K can be associated with several rules k. The functions $\mathcal{C}_i^k$ and $\mathcal{T}_i^k$ are used to compute respectively the context C_i and the term T_i of the ith recursive call to the semantics function (producing the ith subtree PT_i). In general, C_i can depend on the original context C, on previous results R_j and on the subexpression E of the original term.

For instance in the first rule of Figure 4, $C_1 = C$ and $C_2 = R_1$. The evaluation of the terms T_i may involve the subterms U_j of T[1]. For example, in the first rule of Figure 4, $T_1 = U_1$ and $T_2 = U_2$. E is the part of the term that can be used in the computations at the current node of the derivation tree (in the test $\mathcal{P}^k$ or in the computation of the normal form by $\mathcal{F}^k$). For example E is empty for the first rule of Figure 4; it is equal to $Id := Exp$ for the second rule and to Exp for all the other rules. In addition to E, the computation of the normal form by $\mathcal{F}^k$ may involve the normal forms of the subtrees (R_j) and the initial context C. In the first rule of Figure 4, $\mathcal{F}^k(C, (R_1, R_2), E) = R_2$. Note that we do not allow cyclic dependences between the definitions of the C_i, T_i, PT_i and R_i (as usual in natural semantics definitions). We can thus assume without any restriction that the definitions are ordered in such a way that C_i and T_i depend only on R_j with $j < i$. This can be stated more formally:

$$\begin{aligned} \forall i.\ \mathcal{C}_i^k(C, \overline{R}, E) &= \mathcal{C}_i^k(C, \overline{R}[\perp / R_j \ \ j \geq i], E) \\ \forall i.\ \mathcal{T}_i^k(C, \overline{R}, T) &= \mathcal{T}_i^k(C, \overline{R}[\perp / R_j \ \ j \geq i], T) \end{aligned}$$

We characterise the compositionality property mentioned in the introduction by the following assumption:

$$\mathcal{T}_i^k(C, \overline{R}, T[U_k'/U_k]) = \mathcal{T}_i^k(C, \overline{R}, T)[U_k'/U_k]$$

This property expresses the fact that the functions $\mathcal{T}_i^k$ do not decompose the subterms U_k of T but simply use them as part of their result. This constraint excludes functions decomposing the subterms U_k to forge new terms.

We believe that this format captures the way natural semantics definitions are usually presented and does not introduce any real restriction. At least, it is general enough to cover standard first-order sequential programming languages as we will see in sections 4, 5 and 6. The extension to higher-order languages is discussed in the conclusion.

[1] In fact, it is often the case that T_i is equal to one of the U_j, as can be seen from Figure 5.

3 A generic slicing analyser

Slicing[2] a program consists in constructing a reduced version of the program (called a program *slice*) containing only those statements that affect a given set of variables at given program points (this set is called *the slicing criterion*). In program debugging, slicing makes it possible for a software engineer to focus on the relevant parts of the code. Slicing is also useful for testing, understanding and in maintenance activities. Because of this diversity of applications, different variations on the notion of slicing have been proposed, as well as a number of methods to compute slices. First a program slice can either be *executable* or not. Producing an executable slice makes it possible to apply further treatments to the result of the analysis. An other important distinction is between *static* and *dynamic* slicing. In the first case, the slice is computed without any assumption on the inputs, whereas the latter relies on some specific input data. Slicing algorithms can also be divided into two categories: *forward* and *backward* analysers. *Backward* slicing identifies the statements of a program that may have some impact on the criterion whereas *forward* slicing returns the statements which may be influenced by the criterion. In this paper, we consider *dynamic backward slicing* with executable slices. Static slicing algorithms can be derived by abstract interpretation of dynamic slicing analysers [5] ; this construction is sketched in the conclusion. We can describe forward slicing analysers in a similar way but slicing analyses producing non executable slices do not fit well into our framework since the specification of the analysis is a relation between the semantics of the original program and the semantics of the slice.

We consider the following imperative program as an illustration of dynamic backward slicing (some program points are omitted for the sake of readability):

$$
\begin{array}{l}
(\pi_1, m := n);\\
(\pi_2, av := 0);\\
(\pi_3, max := 0);\\
(\pi_4, min := k_{max});\\
(\pi_5, \texttt{while } n > 0 \texttt{ do} \quad (\pi_6, av := av + t[n]);\\
\qquad\qquad\qquad\qquad\quad (\pi_7, min := Min(min, t[n]));\\
\qquad\qquad\qquad\qquad\quad (\pi_8, max := Max(max, t[n]));\\
\qquad\qquad\qquad\qquad\quad (\pi_9, n := n - 1));\\
(\pi_{10}, av := av/(Max(1, m)))
\end{array}
$$

We assume that we are interested only in the value of variable av at point π_{10}. The pair $(\pi_{10}, \{av\})$ is called the slicing criterion. The dynamic slice of the program is extracted for one particular input. For instance, if we execute the program with 0 as the initial value of n, we get:

$$
\begin{array}{l}
(\pi_1, m := n);\\
(\pi_2, av := 0);\\
(\pi_{10}, av := av/(Max(1, m)))
\end{array}
$$

In this case, the body of the loop is never executed. If we consider the execution of the program with 2 as initial value of n, the loop is executed and the dynamic slice for the criterion is the following:

$$
\begin{array}{l}
(\pi_1, m := n);\\
(\pi_2, av := 0);\\
(\pi_5, \texttt{while } n > 0 \texttt{ do} \quad (\pi_6, av := av + t[n]);\\
\qquad\qquad\qquad\qquad\quad (\pi_9, n := n - 1));\\
(\pi_{10}, av := av/(Max(1, m)))
\end{array}
$$

It is easy to check that both slices are still executable programs and that they compute the same values as the original one (for the corresponding initial value of n) for av at point π_{10}.

Slicing was originally proposed by Weiser for imperative languages [23] and its application to logic programming [19] and functional programming [14] have been studied recently. In fact, the concept of slicing itself is very general: it is not tied to one specific style of programming[3] and it can lead to dynamic as well as static analysers [20]. So we believe that slicing is an ideal application to illustrate our framework.

[2] More precisely "backward slicing".

[3] Even if the details of the resulting analyses are of course.

3.1 Formal definition

We only make two assumptions on the domains of the natural semantics of the language here:

- Contexts are functions from program variables to a given domain D of values ($\mathtt{C} = \mathtt{Pvar} \rightarrow \mathtt{D}$).
- Normal forms are themselves contexts ($\mathtt{NF} = \mathtt{C}$).

As we will see later (sections 4, 5, 6), these assumptions make the definition of generic analyses easier without introducing real restrictions on the programming languages[4].

$$
\begin{array}{l}
\mathcal{SL}(C,T,RV,D) = \\
\quad \mathtt{case}\ T\ \mathtt{of} \\
\qquad [(\pi, op^K(E,\overline{U}))\ \mathtt{and}\ \mathcal{P}^k(C,E)\ : \\
\qquad\quad \mathtt{let}\ \ C_i = \mathcal{C}_i^k(C,\overline{R},E) \\
\qquad\qquad\ \ T_i = \mathcal{T}_i^k(C,\overline{R},T) \\
\qquad\qquad\ \ PT_i = \mathcal{S}(C_i,T_i) \\
\qquad\qquad\ \ R_i = PT_i.\mathtt{stt.nf} \\
\qquad\qquad\ \ D' = D + RV(\pi) \\
\qquad\qquad\ \ D_i = +_j\ \mathcal{N}_i(\mathcal{C}_j^k,N_j)\ +_j\ \mathcal{N}'_i(\mathcal{T}_j^k)\ +\ \mathcal{N}_i(\mathcal{F}^k,D') \\
\qquad\qquad\ \ (S_i,N_i) = \mathcal{SL}(C_i,T_i,RV,D_i) \\
\qquad\qquad\ \ N = +_j\ \mathcal{N}_0(\mathcal{C}_j^k,N_j)\ +_j\ \mathcal{N}'_0(\mathcal{T}_j^k)\ +\ \mathcal{N}_0(\mathcal{F}^k,D')\ +\ \mathcal{N}'_0(\mathcal{P}^k) \\
\qquad\quad \mathtt{in}\ \ \mathtt{if}\ RV(\pi) = \emptyset\ \mathtt{and} +_i\ S_i = \emptyset\ \mathtt{and}\ \mathcal{A}(\mathcal{F}^k,D)\ \mathtt{and}_i\ \mathcal{A}(\mathcal{C}_i^k,N_i) \\
\qquad\qquad\ \ \mathtt{then}\ (\emptyset, D\ +\ \mathcal{N}'_0(\mathcal{P}^k)) \\
\qquad\qquad\ \ \mathtt{else}\ (+_i\ S_i + \{\pi\}, N)]^* \\
\\
\quad \mathcal{N}_i(G,N) = N_i \Rightarrow \\
\quad (a'_i =_{N_i} a_i \Rightarrow G(a_0,...,a_{i-1},a'_i,a_{i+1},...,a_k) =_N G(a_0,...,a_{i-1},a_i,a_{i+1},...,a_k)) \\
\quad \mathcal{N}'_i(G) = N_i \Rightarrow \\
\quad (a'_i =_{N_i} a_i \Rightarrow G(a_0,...,a_{i-1},a'_i,a_{i+1},...,a_k) = G(a_0,...,a_{i-1},a_i,a_{i+1},...,a_k)) \\
\quad \mathcal{A}(G,N) \Rightarrow \\
\quad \exists m.G(a_0,a_1,...,a_k) =_N a_m
\end{array}
$$

Figure 6: Slicing: generic analysis

Our generic slicing analyser is presented in Figure 6. The analyser takes the same arguments $C \in \mathtt{C}$ and $T \in \mathtt{T}$ as the semantics function, plus two additional parameters $RV \in \mathtt{PP} \rightarrow \mathcal{P}(\mathtt{Pvar})$ and $D \in \mathcal{P}(\mathtt{Pvar})$. The third argument RV (for Relevant Variables) is the slicing criterion mentioned above. It associates each program point with a set of program variables. A program point π associated with a non-empty set $RV(\pi)$ is called an observation point. In the above example, RV is the function associating π_{10} with $\{av\}$ and all other program points with the empty set:

$$RV(\pi) = \ \mathtt{if}\ \pi = \pi_{10}\ \mathtt{then}\ \{av\}\ \mathtt{else}\ \emptyset$$

The fourth argument D of the analyser represents the set of variables whose value must be preserved in the output context[5] (normal form) of the term. In the initial call D is the empty set. The result of the analysis is a pair (S,N) with $S \in \mathcal{P}(\mathtt{PP})$ and $N \in \mathcal{P}(\mathtt{Pvar})$. S is the set of program points of the term T that must be kept in the slice and N is the set of variables whose value must be preserved in the input context[6]. A program point must be kept in the slice if it can influence an observation point or the value of a variable of D in the output context. The same condition applies to decide which

[4] In a nutshell, a context will be a state for an imperative language, a substitution tree for a logic programming language and an environment for a functional language (with a specific variable denoting the normal form).

[5] For a forward analysis this argument would characterise the input context rather than the output context.

[6] For a forward analysis this argument would characterise the output context rather than the input context.

variables must be preserved in the input context. We use the notation $C =_N C'$ to denote the property $\forall x \in N.\ C\ x = C'\ x$, which means that the restrictions of C and C' to the variables in N are equal.

In order to convey the intuition about the generic definition of Figure 6, we consider its instantiation to the first rule of the semantics of the imperative language presented above (Figure 4). We strictly follow the pattern of Figure 6 for the sake of clarity, avoiding any simplification at this stage. The simplified version is shown in Figure 10.

```
case T of
     (π, U1; U2) :
      let   C1 = C    C2 = R1
            T1 = U1   T2 = U2
            PT1 = S(C1, T1)   PT2 = S(C2, T2)
            R1 = PT1.stt.nf   R2 = PT2.stt.nf
            D' = D + RV(π)
            D1 = N2 + ∅ + ∅   D2 = ∅ + ∅ + D'
            (S1, N1) = SL(C1, T1, RV, D1)   (S2, N2) = SL(C2, T2, RV, D2)
            N = N1 + ∅ + ∅ + ∅
      in    if RV(π) = ∅ and S1 + S2 = ∅ and true and true
            then (∅, D + ∅)
            else ( S1 + S2 + {π}, N)
```

The first lines in the definition of $\mathcal{SL}$ are similar to the definition of the semantics function $\mathcal{S}$. The fact that $\mathcal{SL}$ itself calls $\mathcal{S}$ shows that it is a dynamic analysis. $\mathcal{SL}$ is called recursively on the intermediate contexts and terms of the natural semantics (C_i and T_i). The third argument of the recursive call to $\mathcal{SL}$ is RV, the mapping from program points to relevant variables which is invariant throughout the analysis. The fourth argument D_i represents the set of variables that must be preserved in the result R_i of the evaluation of T_i. D_i is computed using the intermediate functions $\mathcal{N}_i$ and $\mathcal{N}'_i$. $\mathcal{N}_i(G, N)$ returns the set of variables that must be preserved in the $i + 1^{th}$ argument of G for the variables of N to be preserved in the output of G[7]. $\mathcal{N}'_i(G)$ is a specialisation of $\mathcal{N}_i(G, N)$ returning the set of variables that must be preserved in the $i + 1^{th}$ argument of G for the result of G to be unchanged. So, D_i is the sum of all the parts of R_i that are needed by the intermediate functions $\mathcal{C}^k_j$, $\mathcal{T}^k_j$ and $\mathcal{F}^k$ to compute the required part of their result.

In the rule for the sequential command of the imperative language above, $\mathcal{C}^{k_1}_1(C, \overline{R}, E) = C$; thus $\mathcal{N}_0(\mathcal{C}^{k_1}_1, N_1) = N_1$, $\mathcal{N}_1(\mathcal{C}^{k_1}_1, N_1) = \emptyset$ and $\mathcal{N}_2(\mathcal{C}^{k_1}_1, N_1) = \emptyset$. In other words, only the variables of the first argument C can affect the result of $\mathcal{C}^{k_1}_1$. But $\mathcal{C}^{k_1}_2(C, \overline{R}, E) = R_1$ and $\mathcal{N}_2(\mathcal{C}^{k_1}_2, N_2) = N_2$, which means that the set of variables of R_1 which may influence the variables N_2 in the result of $\mathcal{C}^{k_1}_2$ is precisely N_2. The other elements of the sums in the definitions of D_1 and D_2 are computed in the same way and we get $D_1 = N_2$ and $D_2 = D' = D + RV(\pi)$. The intuitive explanation for $D_2 = D + RV(\pi)$ is the following: D_2 is the set of variables whose values must be preserved in R_2; since R_2 is also the result of the whole call to $\mathcal{S}$ (see Figure 4) D_2 must contain D; it must also contain the set $RV(\pi)$ of observation variables at this point. Similarly $D_1 = N_2$ because N_2 is the set of variables that must be preserved in the input context C_2 of the second call to $\mathcal{SL}$ and $C_2 = R_1$, so N_2 is also the set of variables that must be preserved in R_1.

From the test in the definition of $\mathcal{SL}$, we see that three conditions must be met for a program point π to be excluded from the slice:

1. The program point should not be an observation point, that is to say the set of relevant variables $RV(\pi)$ should be empty.

2. No embedded program point should belong to the slice ($+_i\ S_i = \emptyset$).

[7] Both the argument and the output are of type context.

3. No transformation of the input context should occur at this point ($\mathcal{A}(\mathcal{F}^k, D)$ $\texttt{and}_i$ $\mathcal{A}(\mathcal{C}_i^k, N_i)$). The function $\mathcal{A}(G, N)$ returns true if the result of G (restricted to the variables in N) is equal to one of its arguments.

The third condition holds for the rule of the imperative language above (because $\mathcal{F}^{k_1}$, $\mathcal{C}_1^{k_1}$ and $\mathcal{C}_2^{k_1}$ return one of their arguments) and the test reduces to $RV(\pi) = \emptyset$ $\texttt{and}$ $S_1 + S_2 = \emptyset$.

If the program point can be removed from the slice, the result of the analysis is $(\emptyset, D \ + \ \mathcal{N}_0'(\mathcal{P}^k))$, which means that no program point is added to the slice and the variables whose value must be preserved in the input context are the variables that are necessary in the output context plus the variables needed to evaluate the side condition $\mathcal{P}^k(C, E)$. Otherwise, the first component of the result of the analysis is $+_i\ S_i + \{\pi\}$ because π has to be added to the program points collected in the subterms of T. The second component N of the result is the set of variables whose value must be preserved in the input context C. It is computed using $\mathcal{N}_0$ and $\mathcal{N}_0'$ in a way which is very similar to the D_i. Note that $\mathcal{N}_0$ and $\mathcal{N}_0'$ are used rather than $\mathcal{N}_i$ and $\mathcal{N}_i'$ here because C is the first argument of the functions $\mathcal{C}_j^k$, $\mathcal{T}_j^k$, $\mathcal{F}^k$, and $\mathcal{P}^k$. So N is the sum of all the parts of the input context C that are needed by the intermediate functions $\mathcal{C}_j^k$, $\mathcal{T}_j^k$, $\mathcal{F}^k$, and $\mathcal{P}^k$.

In the illustrating rule above $\mathcal{P}^k$ is the constant function **true**, so $\mathcal{N}_0'(\mathcal{P}^k) = \emptyset$ and the first term of the condition reduces to $(\emptyset, D)$.

3.2 Correctness property

As mentioned in the introduction, we propose to specify an analyser by a transformation of the analysed program and a relation that the initial program and the transformed program must satisfy. The transformation is expressed through program annotations. The function $\mathcal{A}nn(S, T)$ returns the annotated term obtained using the set of program points S returned by the slicing analysis:

$$\begin{array}{rcll} \mathcal{A}nn(S, (\pi,\ op^K(E, \overline{U}))) & = & (\pi,\ op^K_{true}(E, \overline{U'})) & \text{if} \quad \pi \in S \\ & = & (\pi,\ op^K_{false}(E, \overline{U'})) & \text{if} \quad \pi \notin S \\ \text{with} \quad U_i' & = & \mathcal{A}nn(S, U_i) & \end{array}$$

The semantics of the language is extended in the following way to deal with annotated operators:

$$\begin{array}{l} \mathcal{P}^k_{true}(C, E) = \mathcal{P}^k(C, E) \\ \mathcal{P}^k_{false}(C, E) = true \\ \mathcal{C}^k_{\theta i}(C, \overline{R}, E) = \mathcal{C}^k_i(C, \overline{R}, E) \quad \text{with} \ \ \theta = true \ \ or \ \ \theta = false \\ \mathcal{T}^k_{\theta i}(C, \overline{R}, T) = \mathcal{T}^k_i(C, \overline{R}, T) \quad \text{with} \ \ \theta = true \ \ or \ \ \theta = false \\ \mathcal{F}^k_{true}(C, \overline{R}, E) = \mathcal{F}^k(C, \overline{R}, E) \\ \mathcal{F}^k_{false}(C, \overline{R}, E) = C \end{array}$$

The intuition is that program points with the annotation *false* are treated as *skip* instructions ($\mathcal{P}^k_{false}(C, E) = true$ and $\mathcal{F}^k_{false}(C, \overline{R}, E) = C$).

As mentioned above, the annotated term derived from the slicing analysis must behave like the original term at the observation points. More precisely, the value of the observation variables $RV(\pi)$ at any point π must be identical in the derivation trees PT and PT' of the original term and its annotated version. This condition is expressed by the relation $\mathcal{R}_b(RV, PT, PT')$:

$$\begin{array}{rcl} \mathcal{R}_b(RV, PT, PT') & = & \forall n \in \text{Nodes}(PT). \\ & & RV(PT[n].\texttt{t.pp}) \ \neq \ \emptyset \Rightarrow \\ & & \quad n \in \text{Nodes}(PT') \ \text{ and} \\ & & \quad \forall x \in RV(PT[n].\texttt{t.pp}).\ \ (PT[n].\texttt{nf})(x) \ = \ (PT'[n].\texttt{nf})(x) \end{array}$$

The notation $PT[n]$ is used to represent the statement at the node n of PT. If (S, N) is the result of the call to the slicing analyser $\mathcal{SL}(C, T, RV, D)$, then we must have $\mathcal{R}_b(RV, \mathcal{S}(C,T), \mathcal{S}(C',T'))$ provided that T' is the annotated term derived from S and T, and C' is equal to C on the variables of N (which is written $C =_N C'$, using the notation introduced earlier). In addition, the values of the variables of D must be preserved in the result (or normal form) of T, so the normal forms must satisfy $\mathcal{S}(C,T).\texttt{stt.nf} =_D \mathcal{S}(C',T').\texttt{stt.nf}$. These conditions lead to the following correctness property for the generic slicing analysis.

DEFINITION **3.1** *Correctness of the generic slicing analysis*

$$\begin{array}{ll} \textit{If} & \mathcal{SL}(C,T,RV,D) = (S,N) \\ & C =_N C' \\ & T' = \mathcal{A}\mathrm{nn}(S,T) \\ \\ \textit{then} & \mathcal{R}_b(RV, \mathcal{S}(C,T), \mathcal{S}(C',T')) \\ & \mathcal{S}(C,T).\texttt{stt.nf} =_D \mathcal{S}(C',T').\texttt{stt.nf} \end{array}$$

The proof of correctness of the generic analysis of Figure 6 can be achieved by a double recurrence on the structure of the natural semantics derivation tree:

1. The first recurrence is on the depth of the derivation tree. The induction hypothesis is that for all $k \in [1, n]$:

$$C_k =_{N_k} C'_k \quad \text{and} \quad T'_k = \mathcal{A}nn(S_k, T_k) \quad \text{implies} \quad \mathcal{R}_b(RV, PT_k, PT'_k) \quad \text{and} \quad R_k =_{D_k} R'_k \tag{1}$$

   and the property to be proven is:

$$\mathcal{R}_b(RV, PT, PT') \quad \text{and} \quad \mathcal{F}^k(C, \overline{R}, E) =_D \mathcal{F}^k_\theta(C', \overline{R}', E) \quad \text{and} \quad \mathcal{P}^k(C,E) \Rightarrow \mathcal{P}^k_\theta(C', E) \tag{2}$$

2. The second recurrence is on the list of subtrees of a node in the natural semantics derivation tree. Its goal is to establish the property

$$\forall\, k.\ C_k =_{N_k} C'_k \text{ and } T'_k = \mathcal{A}nn(S_k, T_k) \tag{3}$$

   which concludes the correctness proof.

It is easy to see that both orderings are well-founded if the natural semantics derivation tree is finite (in other words, we consider the analysis of terminating programs only). The interested reader can find a more detailed account of the proof in Appendix 1.

4 Slicing analysis for an imperative programming language

In order to obtain a slicing analysis for a given programming language, we just have to provide definitions for the functions $\mathcal{N}_i$, $\mathcal{N}'_i$ and $\mathcal{A}$ for each intermediate function of the natural semantics format and to show that they meet the constraints at the bottom of Figure 6. We present in Figure 7 and Figure 8 the appropriate definitions for the imperative language whose semantics was introduced in section 2. The definitions of $\mathcal{C}^k_i$, $\mathcal{T}^k_i$, $\mathcal{F}^k$ and $\mathcal{P}^k$ for this language were listed in Figure 4.

We provide a justification for some of the values of Figure 7 and Figure 8.

- $\mathcal{N}_0(\mathcal{C}^{k_1}_1, N_1) = N_1$ because $\mathcal{C}^{k_1}_1(C, \overline{R}, E) = C$. So, $C =_{N_1} C' \Rightarrow \mathcal{C}^{k_1}_1(C, \overline{R}, E) =_{N_1} \mathcal{C}^{k_1}_1(C', \overline{R}, E)$.

- $\mathcal{N}_0(\mathcal{F}^{k_2}, D') = \texttt{if } Id \notin D' \texttt{ then } D' \texttt{ else } D' - \{Id\} + \{x \in Exp\}$ because

$$\mathcal{F}^{k_2}(C, \overline{R}, Id := Exp) = C[Val(C, Exp)/Id].$$

So, if $Id \notin D'$ then $C =_{D'} C' \Rightarrow \mathcal{F}^{k_2}(C, \overline{R}, Id := Exp) =_{D'} \mathcal{F}^{k_2}(C', \overline{R}, Id := Exp)$.
If $Id \in D'$ and $D" = D' - \{Id\} + \{x \in Exp\}$, then

$$C =_{D"} C' \Rightarrow \mathcal{F}^{k_2}(C, \overline{R}, Id := Exp) =_{D'} \mathcal{F}^{k_2}(C', \overline{R}, Id := Exp).$$

because $D"$ contains all the variables whose value affect the value of Id in the result of the statement.

- $\mathcal{A}(\mathcal{F}^{k_2}, D) = (Id \notin D)$ because the input and the output contexts of the statement only differ in the variable Id.

	$\mathcal{N}_0(\mathcal{C}_1^k, N_1)$	$\mathcal{N}_0(\mathcal{C}_2^k, N_2)$	$\mathcal{N}'_0(\mathcal{T}_1^k)$	$\mathcal{N}'_0(\mathcal{T}_2^k)$	$\mathcal{N}_0(\mathcal{F}^k, D')$	$\mathcal{N}'_0(\mathcal{P}^k)$
k_1	N_1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
k_2					if $Id \notin D'$ then D' else $D' - \{Id\} + \{x \in Exp\}$	$\emptyset$
k_3	N_1		$\emptyset$		$\emptyset$	$\{x \in Exp\}$
k_4	N_1		$\emptyset$		$\emptyset$	$\{x \in Exp\}$
k_5	N_1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{x \in Exp\}$
k_6					D'	$\{x \in Exp\}$

	$\mathcal{N}_1(\mathcal{C}_1^k, N_1)$	$\mathcal{N}_1(\mathcal{C}_2^k, N_2)$	$\mathcal{N}'_1(\mathcal{T}_1^k)$	$\mathcal{N}'_1(\mathcal{T}_2^k)$	$\mathcal{N}_1(\mathcal{F}^k, D')$
k_1	$\emptyset$	N_2	$\emptyset$	$\emptyset$	$\emptyset$
k_3	$\emptyset$		$\emptyset$		D'
k_4	$\emptyset$		$\emptyset$		D'
k_5	$\emptyset$	N_2	$\emptyset$	$\emptyset$	$\emptyset$

	$\mathcal{N}_2(\mathcal{C}_1^k, N_1)$	$\mathcal{N}_2(\mathcal{C}_2^k, N_2)$	$\mathcal{N}'_2(\mathcal{T}_1^k)$	$\mathcal{N}'_2(\mathcal{T}_2^k)$	$\mathcal{N}_2(\mathcal{F}^k, D')$
k_1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	D'
k_5	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	D'

Figure 7: Instantiation of $\mathcal{N}_i$ for the imperative language

	$\mathcal{A}(\mathcal{C}_1^k, N_1)$	$\mathcal{A}(\mathcal{C}_2^k, N_2)$	$\mathcal{A}(\mathcal{F}^k, D)$
k_1	**true**	**true**	**true**
k_2			$Id \notin D$
k_3	**true**		**true**
k_4	**true**		**true**
k_5	**true**	**true**	**true**
k_6			**true**

Figure 8: Instantiation of $\mathcal{A}$ for the imperative language

The other values of Figure 7 and Figure 8 can be explained in a similar way. The empty boxes in the tables correspond to irrelevant functions[8] and lines that contain only empty boxes and $\emptyset$ have been omitted.

The values obtained for D_i, N and the test conditions for the imperative language are gathered in Figure 9. They can be derived from Figure 7 and Figure 8. For example, for k_1 and D_1, we have:

[8]For instance, k_2 is an axiom, so it does not involve $\mathcal{C}_i$ and $\mathcal{T}_i$ functions and $\mathcal{N}_i$ and $\mathcal{N}'_i$ are irrelevant.

$$\begin{aligned} D_1 &= +_j \mathcal{N}_1(\mathcal{C}_j^{k_1}, N_j) \ +_j \ \mathcal{N}_1'(\mathcal{T}_j^{k_1}) \ + \ \mathcal{N}_1(\mathcal{F}^{k_1}, D') \\ &= \mathcal{N}_1(\mathcal{C}_1^{k_1}, N_1) + \mathcal{N}_1(\mathcal{C}_2^{k_1}, N_2) + \mathcal{N}_1'(\mathcal{T}_1^{k_1}) + \mathcal{N}_1'(\mathcal{T}_2^{k_1}) + \mathcal{N}_1(\mathcal{F}^{k_1}, D') \\ &= \emptyset + N_2 + \emptyset + \emptyset + \emptyset \\ &= N_2 \end{aligned}$$

We can use these values to instantiate the generic slicing algorithm of Figure 6 and we get the analyser $\mathcal{SL}$ of Figure 10.

	D_1	D_2	N	$\mathcal{A}(\mathcal{F}^k, D)$ and$_{i\ (i=1,2)}$ $\mathcal{A}(\mathcal{C}_i^k, N_i)$
k_1	N_2	D'	N_1	**true**
k_2			if $Id \notin D'$ then D' else $D' - \{Id\} + \{x \in Exp\}$	$Id \notin D$
k_3	D'		$N_1 + \{x \in Exp\}$	**true**
k_4	D'		$N_1 + \{x \in Exp\}$	**true**
k_5	N_2	D'	$N_1 + \{x \in Exp\}$	**true**
k_6			$D' + \{x \in Exp\}$	**true**

Figure 9: Result of the instantiations for the imperative language

Note that the analyser $\mathcal{SL}$ would be very inefficient if executed without any optimisation because it would entail the reevaluation of an expression like $S(C, U_1)$ in the context $(U_1; U_2); U_3$. Two solutions are possible to remove the source of inefficiency: the first one consists in applying a memoization technique to avoid the reevaluation of $\mathcal{S}$ with identical arguments. The second one is to transform the analyser itself to express it as a function of the semantics proof tree. We then get an analyser $\mathcal{SL}'$ satisfying the following property:

$$\mathcal{SL}'(S(C,T), RV, D) = \mathcal{SL}(C, T, RV, D)$$

This solution, which has been studied in detail in [5], corresponds to the usual two phases implementation of dynamic analyses (generation of the execution trace followed by its analysis).

To conclude this section, Figure 11 shows the results produced by the analyser $\mathcal{SL}$ for the small imperative program introduced at the beginning of section 3. The analyser is first called with an initial context such that $t = []$ and $n = 0$. The slicing criterion RV is $\{(\pi_{10}, av)\}$, which means that we are only interested in the value of av at program point π_{10}. The initial output context is $\emptyset$ because no other program variable has to be preserved in the output of the program. The first call of the analyser is $\mathcal{SL}([]/t, 0/n], T, RV, \emptyset)$ with:

$$\begin{aligned} T &= U; U_{10} \\ U &= U'; U_5 \\ U_{10} &= (\pi_{10}, av := av/(Max(1,m))) \\ U' &= (\pi_1, m := n); (\pi_2, av := 0); (\pi_3, max := 0); (\pi_4, min := k_{max}); \\ U_5 &= (\pi_5, \texttt{while } n > 0 \texttt{ do } \ (\pi_6, av := av + t[n]); \\ &\qquad\qquad\qquad\qquad\quad (\pi_7, min := Min(min, t[n])); \\ &\qquad\qquad\qquad\qquad\quad (\pi_8, max := Max(max, t[n])); \\ &\qquad\qquad\qquad\qquad\quad (\pi_9, n := n - 1)) \end{aligned}$$

Applying the first rule of $\mathcal{SL}$ (Figure 10) the initial call triggers the call:

$$\mathcal{SL}(\mathcal{S}(C,U).\texttt{stt.nf}, U_{10}, RV, \emptyset) = (\{\pi_{10}\}, \{av, m\})$$

The analyser finds that π_{10} is kept in the slice because its statement influences the slicing criterion and the input context is the set containing variables used in the statement. The second call is

$\mathcal{SL}(C, T, RV, D) =$
`case` T `of`

$(\pi, U_1; U_2)$:
`let` $PT_1 = \mathcal{S}(C, U_1)$
$R_1 = PT_1$`.stt.nf`
$D' = D + RV(\pi)$
$(S_2, N_2) = \mathcal{SL}(R_1, U_2, RV, D')$
$(S_1, N_1) = \mathcal{SL}(C, U_1, RV, N_2)$
`in` `if` $RV(\pi) = \emptyset$ `and` $S_1 + S_2 = \emptyset$
`then` $(\emptyset, D)$
`else` $(S_1 + S_2 + \{\pi\}, N_1)$

$(\pi, Id := Exp)$:
`let` $D' = D + RV(\pi)$
$N =$ `if` $Id \notin D'$ `then` D' `else` $D' - \{Id\} + \{x \in Exp\}$
`in` `if` $RV(\pi) = \emptyset$ `and` $Id \notin D$
`then` $(\emptyset, D)$
`else` $(\{\pi\}, N)$

$(\pi,$ `cond` $(Exp, U_1, U_2))$ and $Val(C, Exp) =$ **true** :
`let` $D' = D + RV(\pi)$
$(S_1, N_1) = \mathcal{SL}(C, U_1, RV, D')$
`in` `if` $RV(\pi) = \emptyset$ `and` $S_1 = \emptyset$
`then` $(\emptyset, D + \{x \in Exp\})$
`else` $(S_1 + \{\pi\}, N_1 + \{x \in Exp\})$

$(\pi,$ `cond` $(Exp, U_1, U_2))$ and $Val(C, Exp) =$ **false** :
`let` $D' = D + RV(\pi)$
$(S_1, N_1) = \mathcal{SL}(C, U_2, RV, D')$
`in` `if` $RV(\pi) = \emptyset$ `and` $S_1 = \emptyset$
`then` $(\emptyset, D + \{x \in Exp\})$
`else` $(S_1 + \{\pi\}, N_1 + \{x \in Exp\})$

$(\pi,$ `while` Exp `do` $U_1)$ and $Val(C, Exp) =$ **true** :
`let` $PT_1 = \mathcal{S}(C, U_1)$
$R_1 = PT_1$`.stt.nf`
$D' = D + RV(\pi)$
$(S_2, N_2) = \mathcal{SL}(R_1, T, RV, D')$
$(S_1, N_1) = \mathcal{SL}(C, U_1, RV, N_2)$
`in` `if` $RV(\pi) = \emptyset$ `and` $S_1 + S_2 = \emptyset$
`then` $(\emptyset, D + \{x \in Exp\})$
`else` $(S_1 + S_2 + \{\pi\}, N_1 + \{x \in Exp\})$

$(\pi,$ `while` Exp `do` $U_1)$ and $Val(C, Exp) =$ **false** :
`let` $D' = D + RV(\pi)$
`in` `if` $RV(\pi) = \emptyset$
`then` $(\emptyset, D + \{x \in Exp\})$
`else` $(\{\pi\}, D' + \{x \in Exp\})$

Figure 10: Slicing: dynamic analysis for an imperative language

$\mathcal{SL}(C, U, RV, \{av, m\})$ which leads to:

$$\mathcal{SL}(\mathcal{S}(C, U').\texttt{stt.nf}, U_5, RV, \{av, m\}) = (\emptyset, \{av, m, n\})$$
$$\text{and} \quad \mathcal{SL}(C, U', RV, \{av, m, n\}) = (\{\pi_1, \pi_2\}, \{n\})$$

The analyser drops π_5 because the loop is not executed but the input context is extended with n because this variable is used to evaluate the test of the loop. Program points π_4 and π_3 are also dropped because they do not influence the output context. As a consequence, we have:

$$\mathcal{SL}(C, U, RV, \{av, m\}) = (\{\pi_1, \pi_2\}, \{n\})$$

which leads to:

$$\mathcal{SL}([]/t, 0/n], T, RV, \emptyset) = (\{\pi_1, \pi_2, \pi_{10}\}, \{n\})$$

The second call of the dynamic analyser with $t = [2, 3]$ and $n = 2$ as the initial context produces also the expected slice (see section 3). In this case π_5, π_6 and π_9 are kept in the slice because the program goes through these program points for this particular initial context. The analyser finds that only the variables n and t are useful in the input context.

Source program

$T =$ $(\pi_1, m := n);$
$(\pi_2, av := 0);$
$(\pi_3, max := 0);$
$(\pi_4, min := k_{max});$
$(\pi_5, \texttt{while } n > 0 \texttt{ do } (\pi_6, av := av + t[n]);$
$(\pi_7, min := Min(min, t[n]));$
$(\pi_8, max := Max(max, t[n]));$
$(\pi_9, n := n - 1));$
$(\pi_{10}, av := av/(Max(1, m)))$

Analysis results

$RV = \{(\pi_{10}, av)\}$
$\mathcal{SL}([]/t, 0/n], T, RV, \emptyset) = (\{\pi_1, \pi_2, \pi_{10}\}, \{n\})$
$\mathcal{SL}([2, 3]/t, 2/n], T, RV, \emptyset) = (\{\pi_1, \pi_2, \pi_5, \pi_6, \pi_9, \pi_{10}\}, \{n, t\})$

Figure 11: Slicing: dynamic analysis of a simple imperative program

5 Slicing analysis for a logic programming language

The operational semantics of a simple logic programming language is presented in Figure 12. We assume a program *Prog* which is a collection of predicate definitions of the form $[P_k(x_1, ..., x_n) = B_k]$. The body B_k of a predicate is in normal form and it contains only variables from $\{x_1, ..., x_n\}$. Normal forms are first order formulae (also called "goal formulae" in [9]) built up from predicate applications using only the connectives "or" "and", and "there exists". Their syntax is the following (we use the notation `Op` for basic predicates[9] and P_k for user-defined predicates):

$$\texttt{Op}(x_1, x_2, x_3) \mid x = t \mid U_1 \vee U_2 \mid U_1 \wedge U_2 \mid \exists x.U_1 \mid P_k(y_1, \ldots, y_n)$$

[9]We consider only ternary basic predicates here, but other arities are treated in the same way.

We assume that each variable x occurring in a term $\exists x.U_1$ is unique. In a program, each subterm in this syntax is associated with a program point. As an illustration of this syntax, Figure 15 presents a version of the small program introduced in section 3 in a standard logic programming syntax and shows its translation into normal form.

The domain of contexts for this language is defined as follows:

$$\begin{array}{lcl} \texttt{C} & = & \texttt{Tree(Subst)} \\ \texttt{Subst} & = & \texttt{Pvar} \rightarrow \texttt{Rterm} \\ \texttt{Rsubst} & = & \texttt{Rvar} \rightarrow \texttt{Rterm} \end{array}$$

where $\texttt{Tree}(H)$ is the type of binary trees with leaves of type H.

We define contexts as binary trees of substitutions to take into account the non deterministic nature of the language. So we gather in one derivation the computation of all the substitutions of a program. A particular control strategy for the implementation of the language corresponds to a particular ordering of the leaves of substitutions trees. For instance, the list of results of the usual depth-first evaluation strategy of Prolog is precisely the leaves of the substitution tree produced by our semantics ordered from left to right.

Note that the domain $\texttt{C} = \texttt{Tree(Subst)}$ is isomorphic to a restriction of $\texttt{Pvar} \rightarrow \texttt{Tree}(\texttt{Rterm})$, so our definition of C fits with the constraint of section 3.1. Following [7], we assume an infinite set of renaming variables Rvar and we distinguish two kinds of substitutions: program variables substitutions (Psubst) and renaming variables substitution (Rsubst). By convention, we use $\theta \in \texttt{Subst}$ for a program variable substitution and $\sigma \in \texttt{Rsubst}$ for a renaming variable substitution. We write $N(T_1, T_2)$ a tree with subtrees T_1 and T_2.

In Figure 12, $\overline{F}(T)$ denotes the application of a function F to all the substitutions of a tree T. The function $union(T_1, T_2, T_3)$ builds a new substitution tree joining the trees T_2 and T_3 produced by two subgoals. Its first argument is the initial substitution, which is used to identify the points where the joins have to be introduced (these points are the leaves of T_1). *Add* is used to add a program variable in a substitution (the new program variable is attached to a free renaming variable) and *Drop* removes a variable from a substitution. Ren_k creates a new substitution to execute the body of a clause (it amounts to a variable renaming) and Ext_k propagates the result of a predicate in the calling substitutions. From the definition of Ren_k, we see that the body B_k of a predicate is evaluated in an environment defining exactly the formal parameters of the predicate P_k.

Figure 13 presents the semantics of Figure 12 in our format and Figure 14 shows the dynamic analyser obtained by instantiating the generic analyser of Figure 6. The set of definitions *Prog* is used as an implicit parameter of the semantics and the analyser. In order to make it easier to understand why the instantiation of the generic slicing analyser leads to the analyser of Figure 14, we present in Appendix 2 all the instantiation tables of the semantics and the analyser for the logic programming language. Rather than going through these tables to show that the intermediate functions of the analyser satisfy the requirements of Figure 6, we try to convey some intuition about the analysis resulting from this instantiation.

In the definition of $\mathcal{SL}$ in Figure 14, the first line was added to factorise the definitions $\texttt{let}\ D' = D + RV(\pi)$ in each case of the generic slicing analysis. The relation $Indep(C, D_1, D_2)$ is used to ensure that two sets of variables D_1 and D_2 are independent, which is the case when they do not share any renaming variable (in any substitution of the context C). *Indep* appears in the first two cases as a necessary condition to exclude the term from the slice. If the relation holds, then the (renaming variable) substitution resulting from the evaluation of the term cannot have any impact on the variables of D. The relation $UF(C, x, t)$ is satisfied if the unification of x and t cannot fail for any substitution of C. It is a prerequisite for excluding $\texttt{Eq}(x, t)$ from the slice because a failure is recorded in the substitution tree as the $\perp$ substitution[10]; as a consequence, it has an impact on all the variables. This

[10]Note that $\perp$ is an absorbing element for the semantics of the language. For instance $op(\perp, x_1, x_2, x_3) = \perp$ and $unif(\perp, x, t) = \perp$.

$$[k_1] \qquad C \vdash (\pi, \mathtt{Op}(x_1, x_2, x_3)) \rightarrow \overline{op}(C, x_1, x_2, x_3)$$

$$[k_2] \qquad C \vdash (\pi, x = t) \rightarrow \overline{unif}\,(C, x, t)$$

$$[k_3] \quad \frac{C \vdash U_1 \rightarrow R_1 \quad R_1 \vdash U_2 \rightarrow R_2}{C \vdash (\pi, U_1 \wedge U_2) \rightarrow R_2} \qquad [k_4] \quad \frac{C \vdash U_1 \rightarrow R_1 \quad C \vdash U_2 \rightarrow R_2}{C \vdash (\pi, U_1 \vee U_2) \rightarrow \mathit{union}\,(C, R_1, R_2)}$$

$$[k_5] \quad \frac{\overline{Add}\ (C, x, rx) \vdash U_1 \rightarrow R_1}{C \vdash (\pi, \exists x.U_1) \rightarrow \overline{Drop}\ (R_1, x)} \quad rx \in Rvar \quad \text{fresh variable}$$

$$[k_6] \quad \frac{\overline{Ren}_k(C) \vdash B_k \rightarrow R_1}{C \vdash (\pi, P_k(y_1, ..., y_n)) \rightarrow \overline{Ext}_k(C, R_1)} \quad \text{with } [P_k(x_1, ..., x_n) = B_k] \in Prog$$

$\overline{F}(N(T_1, T_2), x_1, \ldots, x_n) = N(\overline{F}(T_1, x_1, \ldots, x_n), \overline{F}(T_2, x_1, \ldots, x_n))$
$\overline{F}(\theta, x_1, \ldots, x_n) = F(\theta, x_1, \ldots, x_n)$

$op(\theta, x_1, x_2, x_3) = \mathtt{let}\ \sigma = [(\theta(x_1)\ \mathrm{op}\ \theta(x_2))/\theta(x_3)]\ \mathtt{in}\ \sigma \circ \theta$
if $\theta(x_1)$ and $\theta(x_2)$ are ground and $\theta(x_3) \in \mathtt{Rvar}$, $\quad \perp$ otherwise

$unif(\theta, x, t) = \quad \mathtt{let}\ \sigma = \mathrm{mgu}(\theta(x), \theta(t))\ \mathtt{in}\ \sigma \circ \theta \quad$ if $\theta(x)$ and $\theta(t)$ can be unified ,
$\perp$ otherwise

$union(N(T_1, T_2),\ N(U_1, U_2),\ N(V_1, V_2)) = N(union(T_1, U_1, V_1),\ union(T_2, U_2, V_2))$
$union(\theta, U, V) = N(U, V)$

$Add(\theta, pv, rv) = \theta[rv/pv]$ with $v \neq pv \Rightarrow \theta[rv/pv](v) = \theta(v)$ and $\theta[rv/pv](pv) = rv$

$Drop(\theta, pv) = \theta_{/pv}$ with $v \neq pv \Rightarrow \theta_{/pv}(v) = \theta(v)$ and $\theta_{/pv}(pv) = \perp$

$Ren_k(\theta) = [\theta(y_i)/x_i]$

$Ext_k(\theta, \theta') = \sigma \circ \theta$ with $\theta' = \sigma \circ [\theta(y_i)/x_i]$

Figure 12: Natural semantics of a logic programming language

$$
\begin{array}{l}
\mathcal{S}(C,T) = \texttt{case}\ \ T\ \texttt{of} \\
\qquad (\pi, \texttt{Op}(x_1,x_2,x_3)) : ((C,T,\overline{op}(C,x_1,x_2,x_3)),\texttt{nil},k_1) \\
\qquad (\pi, \texttt{Eq}(x,t)) : ((C,T,\overline{unif}\,(C,x,t)),\texttt{nil},k_2) \\
\qquad (\pi, \texttt{And}\ (U_1,U_2)) : \ \texttt{let}\quad PT_1 = \mathcal{S}(C,U_1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad R_1 = PT_1.\texttt{stt.nf} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad PT_2 = \mathcal{S}(R_1,U_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad R_2 = PT_2.\texttt{stt.nf} \\
\qquad\qquad\qquad\qquad\qquad\ \ \texttt{in}\quad ((C,T,R_2),[PT_1,PT_2],k_3) \\
\qquad (\pi, \texttt{Or}\ (U_1,U_2)) : \ \texttt{let}\quad PT_1 = \mathcal{S}(C,U_1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad R_1 = PT_1.\texttt{stt.nf} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad PT_2 = \mathcal{S}(C,U_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad R_2 = PT_2.\texttt{stt.nf} \\
\qquad\qquad\qquad\qquad\qquad\ \ \texttt{in}\quad ((C,T,union\,(C,R_1,R_2)),[PT_1,PT_2],k_4) \\
\qquad (\pi, \texttt{Exists}\ (x,U_1)) : \ \texttt{let}\quad PT_1 = \mathcal{S}(\overline{Add}\ (C,x,rx),U_1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad R_1 = PT_1.\texttt{stt.nf} \\
\qquad\qquad\qquad\qquad\qquad\ \ \texttt{in}\quad ((C,T,\overline{Drop}\ (R_1,x)),[PT_1],k_5) \\
\qquad (\pi, \texttt{Call}(P_k(y_1,\ldots,y_n))) : \ \texttt{let}\quad PT_1 = \mathcal{S}(\overline{Ren_k}(C),B_k) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad R_1 = PT_1.\texttt{stt.nf} \\
\qquad\qquad\qquad\qquad\qquad\ \ \texttt{in}\quad ((C,T,\overline{Ext_k}(C,R_1)),[PT_1],k_6) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{with } [P_k(x_1,\ldots,x_n) = B_k] \in \text{Prog}
\end{array}
$$

Figure 13: Semantics of a logic programming language in the natural semantics format

condition was not included in the `Op` case, assuming that the logic programming language is equipped with mode annotations ensuring that operators are always called with their first two arguments ground and the last one free[11]. In both the `Op` and the `Eq` cases, the set of necessary variables (at the input of the program point) is D' added to all the variables of the term.

The rule for `And` is similar to the rule for the sequential operator ";" in the imperative language. The treatment is `Or` is similar except that the term is systematically kept in the slice because it always influences the values of all the variables (through the introduction of subtrees in the derivation tree).

The rules for `Exists` and `Call` are not surprising. We assume that the variable x in $\texttt{Exists}(x,U_1)$ is unique in a normalised program; so x can be removed from the set of necessary variables yielded by the analysis of U_1 (hence $N_1 - \{x\}$). The test in the rule for `Call` is similar to the test in the `Op` case (the origin of the additional condition on N_1 can be found in the definition of $\mathcal{A}(\mathcal{C}_1^{k_6}, N_1)$ in Appendix 2).

The small example used in the previous section to illustrate the slicing of an imperative program can be rewritten in a logic programming language. Its text and results of the slicing analysis are presented in Figure 15. Again, we omit some program points for the sake of readability. The first result of the analysis corresponds to the execution of the program with $Q(nil,x,y,z)$ as the initial call; the slice includes only the body of Q because the third clause of P is never executed[12]. The second result corresponds to a call with $Q((2,(3,nil)),x,y,z)$ and the slice includes, as expected, all the program except the points π_4 and π_5.

To conclude this section, we should stress that our goal here is to illustrate the genericity of our framework rather than to strive for powerful analyses. So, we contented ourselves with a direct application of our format using simple instantiations. More precise dynamic analysers can be obtained

[11] Otherwise an extra condition based on *UF* can be added as in the `Eq` case.

[12] More precisely, in our semantics the third clause is always evaluated with $\perp$ as the input substitution. The definition of *Indep* shows that $Indep(\perp, D_1, D_2)$ is always *true*.

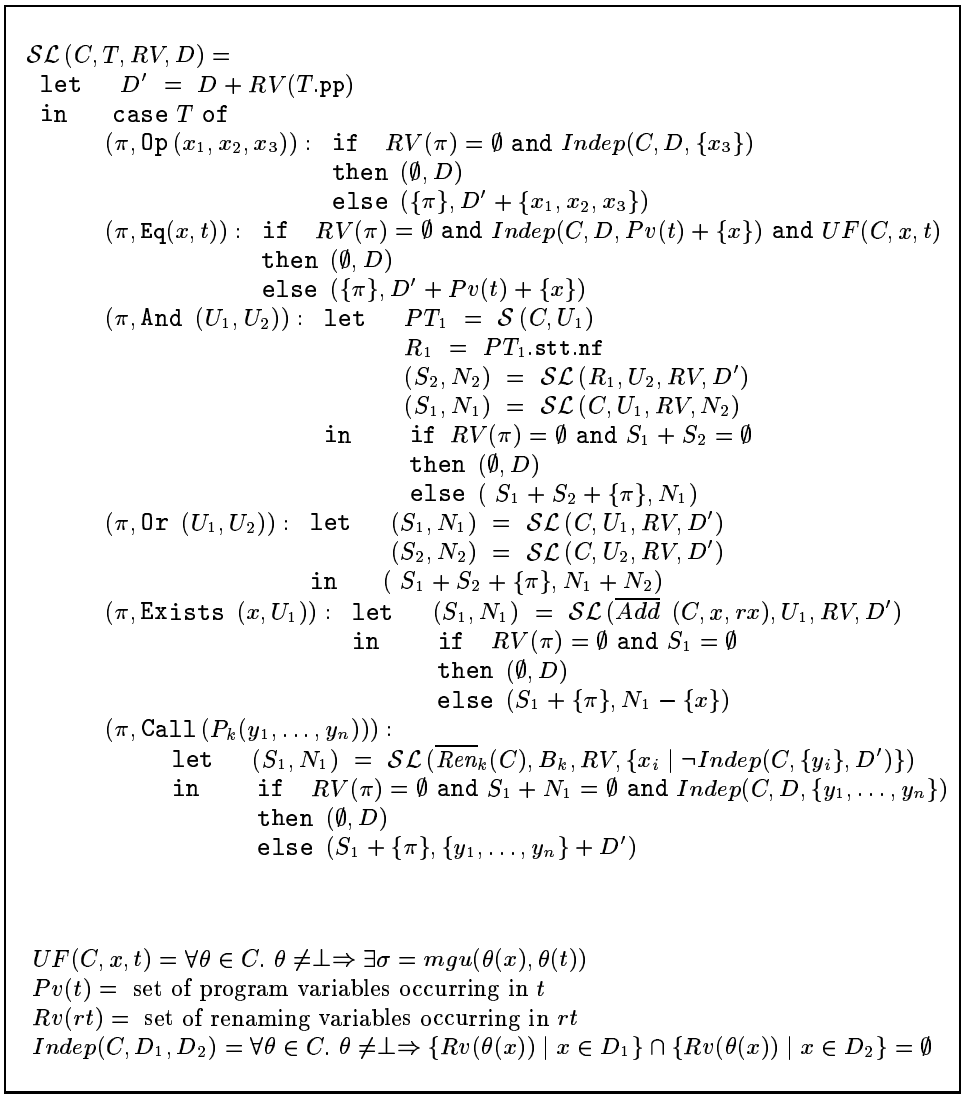

Figure 14: Slicing: dynamic analysis for the logic programming language

Definition of the program in a standard logic programming syntax

$$
\begin{array}{l}
P(nil, 1, 0, 0, 0) \\
P((x, nil), 1, x, x, x) \\
P((x, xs), n, sum, max, min) = \quad (\pi_1, P(xs, n', sum', max', min')) \\
\qquad\qquad\qquad\qquad\qquad\quad (\pi_2, Ad(n', 1, n)) \\
\qquad\qquad\qquad\qquad\qquad\quad (\pi_3, Ad(sum', x, sum)) \\
\qquad\qquad\qquad\qquad\qquad\quad (\pi_4, Max(max', x, max)) \\
\qquad\qquad\qquad\qquad\qquad\quad (\pi_5, Min(min', x, min)) \\
Q(l, av, max, min) = \quad (\pi_6, P(l, n, sum, max, min)) \\
\qquad\qquad\qquad\quad (\pi_7, Div(sum, n, av))
\end{array}
$$

Normal form of the program

$$
\begin{array}{l}
P(l, n, sum, max, min) = \\
\qquad ((l = nil) \wedge (n = 1) \wedge (sum = 0) \wedge (max = 0) \wedge (min = 0)) \vee \\
\qquad (\exists x.\ (l = (x, nil)) \wedge (n = 1) \wedge (sum = x) \wedge (max = x) \wedge (min = x)) \vee \\
\qquad (\exists x.\ \exists xs.\ \exists n'.\ \exists sum'.\ \exists max'.\ \exists min'.\ l = (x, xs) \wedge \\
\qquad\quad (\pi_1, P(xs, n', sum', max', min')) \wedge \\
\qquad\quad (\pi_2, Ad(n', 1, n)) \wedge \\
\qquad\quad (\pi_3, Ad(sum', x, sum)) \wedge \\
\qquad\quad (\pi_4, Max(max', x, max)) \wedge \\
\qquad\quad (\pi_5, Min(min', x, min))) \\
Q(l, av, max, min) = \quad (\exists n.\ \exists sum. \\
\qquad\qquad (\pi_6, P(l, n, sum, max, min)) \wedge \\
\qquad\qquad (\pi_7, Div(sum, n, av)))
\end{array}
$$

Analysis results

$$
\begin{array}{l}
T = Q(l, av, max, min) \\
RV = \{(\pi_7, av)\} \\
\mathcal{SL}\,([nil/l, x/av, y/max, z/min], T, RV, \emptyset) = \\
(\{\pi_6, \pi_7\}, \{l, av, max, min\}) \\
\mathcal{SL}\,([(2, (3, nil))/l, x/av, y/max, z/min], T, RV, \emptyset) = \\
(\{\pi_1, \pi_2, \pi_3, \pi_6, \pi_7\}, \{l, av, max, min\})
\end{array}
$$

Figure 15: Slicing: dynamic analysis of a simple logic program

either through more sophisticated choices for the instantiation functions of Appendix 2 (for instance, in the rule for `Call`, $\mathcal{N}_0(\mathcal{F}^k, D')$ can be improved to avoid the inclusion of all the $y_1, \ldots, y_n$ in the set of necessary variables) or through transformations of the analyser obtained by the direct instantiation (for instance, it can be shown that the condition $N_1 = \emptyset$ in the last case is redundant).

6 Slicing analysis for a functional programming language

The syntax of a simple (call by value) first-order functional language is defined in Figure 16. The semantics of the language in terms of inference rules is presented in Figure 17; Figure 18 shows its definition in our format. In order to meet the constraint that the domain of contexts and the domain of normal forms are identical, we use a special variable res in the environment to denote the result of an expression. The normal form of an expression is the initial environment extended with the variable res. Apart from that, the semantics of Figure 18 is fairly standard. As in the section for an imperative language, a function Val is used to evaluate the value of an operator for a variable in current environment. As in the previous section, we assume a program $Prog$ containing a collection of function declarations of the form $[F(x_1, \ldots, x_n) = E]$. We give just the rule for binary functions here, but n-ary functions are treated similarly. In the same way, we use the [ ] constructor to represent tuples of expressions, but we only give the rule for pairs. The language features `let` expressions to extract the components of a tuple of elements.

Providing the appropriate definitions for the intermediate functions and instantiating the generic analyser of Figure 6, we obtain the dynamic slicing analyser of Figure 19. The details of the instantiation can be found in Appendix 3. We just provide some intuition about the rules of the dynamic analysis here. The only particularity of this definition is the use of variables res_i as a shorthand notation to access the different fields of a tuple result. The condition $D?res$, which occurs in most of the rules, is true if the value of res is not necessary in the result of the expression (and none of its parts res_i either). It is a prerequisite for removing an expression from the slice. The other conditions are similar to the tests appearing in the analysers of section 4 and section 5. In the second part of the result of $\mathcal{SL}$, D'/res is used to remove res (and all its parts res_i) from the set of variables that are necessary in the input context. This is because res is always associated with a new value in the output context. In addition, res is never defined in the input context. The set D_i that is passed as the last argument of the recursive calls to $\mathcal{SL}$ is a subset of $\{res, res_1, \ldots\}$. It represents the necessary part of the result of the corresponding subterm. This demand is propagated backwards from the argument D of the main call. For instance, in the rule for $[U_1, U_2]$, this set is empty for U_1 if $res.1$ is not in D', otherwise it is equal to $\{res\}$. In the same way, in the rule for $F(U_1, U_2)$, the set is empty for U_1 if the variable x is not necessary for the evaluation of E (with $F(x, y) = E$), otherwise it is equal to {res}.

The small example used in the previous sections can be expressed in the functional language and processed by the dynamic analyser. The text of the program and the results of the analyses are shown in Figure 20.

$$
\begin{array}{lcl}
T & ::= & \texttt{Const}\ b \mid \texttt{Var}\ x \mid \texttt{Op}\ (U_1, U_2) \mid [U_1, U_2] \\
 & & \texttt{Cond}\ \ (Op(x), U_1, U_2) \mid F(U_1, U_2) \mid \texttt{let}\ (x_1, x_2)\ =\ U_1\ \texttt{in}\ U_2 \\
 & & \text{with } F(x, y) = E \\
b & ::= & \textbf{true} \mid \textbf{false} \mid n
\end{array}
$$

Figure 16: Syntax of a functional language

$$
\begin{array}{ll}
[k_1] & C \vdash \texttt{Const}\ b \rightarrow \texttt{Const}\ b \\
[k_2] & C \vdash \texttt{Var}\ x \rightarrow C(x) \\
[k_3] & \dfrac{C \vdash U_1 \rightarrow R_1 \quad C \vdash U_2 \rightarrow R_2}{C \vdash \texttt{Op}\ (U_1, U_2) \rightarrow op(R_1, R_2)} \\
[k_4] & \dfrac{C \vdash U_1 \rightarrow R_1 \quad C \vdash U_2 \rightarrow R_2}{C \vdash [U_1, U_2] \rightarrow (R_1, R_2)} \\
[k_5] & \dfrac{C \vdash U_1 \rightarrow R_1}{C \vdash \texttt{Cond}\ \ (Op(x), U_1, U_2) \rightarrow R_1} \quad op(C, x) = \textbf{true} \\
[k_6] & \dfrac{C \vdash U_2 \rightarrow R_2}{C \vdash \texttt{Cond}\ \ (Op(x), U_1, U_2) \rightarrow R_2} \quad op(C, x) = \textbf{false} \\
[k_7] & \dfrac{C \vdash U_1 \rightarrow R_1 \quad C \vdash U_2 \rightarrow R_2 \quad [R_1/x, R_2/y] \vdash E \rightarrow R_3}{C \vdash F(U_1, U_2) \rightarrow R_3} \quad F(x, y) = E \in Prog \\
[k_8] & \dfrac{C \vdash U_1 \rightarrow (v_1, v_2) \quad C[v_1/x_1, v_2/x_2] \vdash U_2 \rightarrow R_2}{C \vdash \texttt{let}\ (x_1, x_2)\ =\ U_1\ \texttt{in}\ U_2 \rightarrow R_2}
\end{array}
$$

Figure 17: Semantics of a functional language

$\mathcal{S}\,(C,T) = \texttt{case}\ \ T\ \texttt{of}$

$(\pi, \texttt{Const}\ b)$:
$\quad ((C,T,C[b/res]), \texttt{nil}, k_1)$

$(\pi, \texttt{Var}\ x)$:
$\quad ((C,T,C[C(x)/res]), \texttt{nil}, k_2)$

$(\pi, \texttt{Op}(U_1,U_2))$:
$\texttt{let}\quad PT_1 = \mathcal{S}\,(C,U_1)$
$\qquad\ R_1 = PT_1.\texttt{stt.nf}$
$\qquad\ PT_2 = \mathcal{S}\,(C,U_2)$
$\qquad\ R_2 = PT_2.\texttt{stt.nf}$
$\qquad\ v = op(R_1(res), R_2(res))$
$\texttt{in}\quad ((C,T,C[v/res]), [PT_1, PT_2], k_3)$

$(\pi, [U_1,U_2])$:
$\texttt{let}\quad PT_1 = \mathcal{S}\,(C,U_1)$
$\qquad\ R_1 = PT_1.\texttt{stt.nf}$
$\qquad\ PT_2 = \mathcal{S}\,(C,U_2)$
$\qquad\ R_2 = PT_2.\texttt{stt.nf}$
$\qquad\ v = [R_1(res), R_2(res)]$
$\texttt{in}\quad ((C,T,C[v/res]), [PT_1, PT_2], k_4)$

$(\pi, \texttt{cond}\ (Op(x), U_1, U_2))\ \texttt{and}\ Val(op)(C,x) = \textbf{true}$:
$\texttt{let}\quad PT_1 = \mathcal{S}\,(C,U_1)$
$\qquad\ R_1 = PT_1.\texttt{stt.nf}$
$\texttt{in}\quad ((C,T,R_1), [PT_1], k_5)$

$(\pi, \texttt{cond}\ (Op(x), U_1, U_2))\ \texttt{and}\ Val(op)(C,x) = \textbf{false}$:
$\texttt{let}\quad PT_1 = \mathcal{S}\,(C,U_2)$
$\qquad\ R_1 = PT_1.\texttt{stt.nf}$
$\texttt{in}\quad ((C,T,R_1), [PT_1], k_6)$

$(\pi, F(U_1,U_2))$:
$\texttt{let}\quad PT_1 = \mathcal{S}\,(C,U_1)$
$\qquad\ R_1 = PT_1.\texttt{stt.nf}$
$\qquad\ PT_2 = \mathcal{S}\,(C,U_2)$
$\qquad\ R_2 = PT_2.\texttt{stt.nf}$
$\qquad\ PT_3 = \mathcal{S}\,([R_1(res)/x,\ R_2(res)/y], E)$
$\qquad\ R_3 = PT_3.\texttt{stt.nf}$
$\texttt{in}\quad ((C,T,C[R_3(res)/res]), [PT_1, PT_2, PT_3], k_7)$
$\quad \text{with}[F(x,y) = E] \in Prog$

$(\pi, \texttt{let}\ [x_1,x_2] = U_1\ \texttt{in}\ U_2)$:
$\texttt{let}\quad PT_1 = \mathcal{S}\,(C,U_1)$
$\qquad\ R_1 = PT_1.\texttt{stt.nf}$
$\qquad\ [v_1,v_2] = R_1(res)$
$\qquad\ PT_2 = \mathcal{S}\,(C[v_1/x_1,\ v_2/x_2], U_2)$
$\qquad\ R_2 = PT_2.\texttt{stt.nf}$
$\texttt{in}\quad ((C,T,C[R_2(res)/res]), [PT_1, PT_2], k_8)$

Figure 18: Semantics of a functional language in the natural semantics format

$$
\begin{array}{l}
\mathcal{SL}\,(C,T,RV,D) = \\
\quad \texttt{let} \quad D' \;=\; D + RV(T.\texttt{pp}) \\
\quad \texttt{in} \quad \texttt{case}\ T\ \texttt{of} \\
\qquad (\pi, \texttt{Const}\ b) : \\
\qquad\quad \texttt{if}\ RV(\pi) = \emptyset\ \texttt{and}\ D?res\ \texttt{then}\ (\emptyset, D)\ \texttt{else}\ (\{\pi\}, D'/res) \\
\qquad (\pi, \texttt{Var}\ x) : \\
\qquad\quad \texttt{if}\ RV(\pi) = \emptyset\ \texttt{and}\ D?res\ \texttt{then}\ (\emptyset, D)\ \texttt{else}\ (\{\pi\}, D'/res + \{x\}) \\
\qquad (\pi, \texttt{Op}(U_1, U_2)) : \\
\qquad\quad \texttt{let} \quad (S_1, N_1) \;=\; \mathcal{SL}\,(C, U_1, RV, \{res\} \cap D') \\
\qquad\qquad\quad\ \ (S_2, N_2) \;=\; \mathcal{SL}\,(C, U_2, RV, \{res\} \cap D') \\
\qquad\quad \texttt{in} \quad \texttt{if}\ RV(\pi) = \emptyset\ \texttt{and}\ D?res\ \texttt{and}\ S_1 + S_2 = \emptyset \\
\qquad\qquad\quad\ \ \texttt{then}\ (\emptyset, D) \\
\qquad\qquad\quad\ \ \texttt{else}\ (S_1 + S_2 + \{\pi\}, N_1 + N_2 + D'/res) \\
\qquad (\pi, [U_1, U_2]) : \\
\qquad\quad \texttt{let} \quad (S_1, N_1) \;=\; \mathcal{SL}\,(C, U_1, RV, \texttt{if}\ res.1 \notin D'\ \texttt{then}\ \emptyset\ \texttt{else}\ \{res\}) \\
\qquad\qquad\quad\ \ (S_2, N_2) \;=\; \mathcal{SL}\,(C, U_2, RV, \texttt{if}\ res.2 \notin D'\ \texttt{then}\ \emptyset\ \texttt{else}\ \{res\}) \\
\qquad\quad \texttt{in} \quad \texttt{if}\ RV(\pi) = \emptyset\ \texttt{and}\ D?res\ \texttt{and}\ S_1 + S_2 = \emptyset \\
\qquad\qquad\quad\ \ \texttt{then}\ (\emptyset, D) \\
\qquad\qquad\quad\ \ \texttt{else}\ (S_1 + S_2 + \{\pi\}, N_1 + N_2 + D'/res) \\
\qquad (\pi, \texttt{cond}\ (Op(x), U_1, U_2))\ \texttt{and}\ Val(op)(C, x) = \textbf{true} : \\
\qquad\quad \texttt{let} \quad (S_1, N_1) \;=\; \mathcal{SL}\,(C, U_1, RV, D') \\
\qquad\quad \texttt{in} \quad \texttt{if}\ RV(\pi) = \emptyset\ \texttt{and}\ S_1 = \emptyset \\
\qquad\qquad\quad\ \ \texttt{then}\ (\emptyset, D + \{x\}) \\
\qquad\qquad\quad\ \ \texttt{else}\ (S_1 + \{\pi\}, N_1 + \{x\}) \\
\qquad (\pi, \texttt{cond}\ (Op(x), U_1, U_2))\ \texttt{and}\ Val(op)(C, x) = \textbf{false} : \\
\qquad\quad \texttt{let} \quad (S_1, N_1) \;=\; \mathcal{SL}\,(C, U_2, RV, D') \\
\qquad\quad \texttt{in} \quad \texttt{if}\ RV(\pi) = \emptyset\ \texttt{and}\ S_1 = \emptyset \\
\qquad\qquad\quad\ \ \texttt{then}\ (\emptyset, D + \{x\}) \\
\qquad\qquad\quad\ \ \texttt{else}\ (S_1 + \{\pi\}, N_1 + \{x\}) \\
\qquad (\pi, F(U_1, U_2)) : \\
\qquad\quad \texttt{let} \quad PT_1 \;=\; \mathcal{S}\,(C, U_1) \\
\qquad\qquad\quad\ \ R_1 \;=\; PT_1.\texttt{stt.nf} \\
\qquad\qquad\quad\ \ PT_2 \;=\; \mathcal{S}\,(C, U_2) \\
\qquad\qquad\quad\ \ R_2 \;=\; PT_2.\texttt{stt.nf} \\
\qquad\qquad\quad\ \ (S_3, N_3) \;=\; \mathcal{SL}\,([R_1(res)/x, R_2(res)/y], E, RV, D'!res) \\
\qquad\qquad\quad\ \ (S_1, N_1) \;=\; \mathcal{SL}\,(C, U_1, RV, \texttt{if}\ x \notin N_3\ \texttt{then}\ \emptyset\ \texttt{else}\ \{res\}) \\
\qquad\qquad\quad\ \ (S_2, N_2) \;=\; \mathcal{SL}\,(C, U_2, RV, \texttt{if}\ y \notin N_3\ \texttt{then}\ \emptyset\ \texttt{else}\ \{res\}) \\
\qquad\quad \texttt{in} \quad \texttt{if}\ RV(\pi) = \emptyset\ \texttt{and}\ D?res\ \texttt{and}(\{x, y\} \cap N_3 = \emptyset)\ \texttt{and}\ S_1 + S_2 + S_3 = \emptyset \\
\qquad\qquad\quad\ \ \texttt{then}\ (\emptyset, D) \\
\qquad\qquad\quad\ \ \texttt{else}\ (S_1 + S_2 + S_3 + \{\pi\}, N_1 + N_2 + D'/res) \\
\qquad (\pi, \texttt{let}\ [x_1, x_2] = U_1\ \texttt{in}\ U_2) : \\
\qquad\quad \texttt{let} \quad PT_1 \;=\; \mathcal{S}\,(C, U_1) \\
\qquad\qquad\quad\ \ R_1 \;=\; PT_1.\texttt{stt.nf} \\
\qquad\qquad\quad\ \ [v_1, v_2] \;=\; R_1(res) \\
\qquad\qquad\quad\ \ (S_2, N_2) \;=\; \mathcal{SL}\,(C[v_1/x_1,\ v_2/x_2], U_2, RV, D'!res) \\
\qquad\qquad\quad\ \ D_1 = \quad (\texttt{if}\ x_1 \notin N_2\ \texttt{then}\ \emptyset\ \texttt{else}\ \{res.1\}) + \\
\qquad\qquad\qquad\qquad\ \ (\texttt{if}\ x_2 \notin N_2\ \texttt{then}\ \emptyset\ \texttt{else}\ \{res.2\}) \\
\qquad\qquad\quad\ \ (S_1, N_1) \;=\; \mathcal{SL}\,(C, U_1, RV, D_1) \\
\qquad\quad \texttt{in} \quad \texttt{if}\ RV(\pi) = \emptyset\ \texttt{and}\ D?res\ \texttt{and}\ (\{x_1, x_2\} \cap N_2 = \emptyset)\ \texttt{and}\ S_1 + S_2 = \emptyset \\
\qquad\qquad\quad\ \ \texttt{then}\ (\emptyset, D) \\
\qquad\qquad\quad\ \ \texttt{else}\ (S_1 + S_2 + \{\pi\}, N_1 + (N_2 - \{x_1, x_2\}) + D'/res)
\end{array}
$$

$$
\begin{array}{l}
D?res = (D \cap \{res, res_1, \ldots\} = \emptyset) \\
D'/res = D' - \{res, res_1, \ldots\} \\
D'!res = D' \cap \{res, res_1, \ldots\}
\end{array}
$$

Figure 19: Slicing: dynamic analysis for the functional language

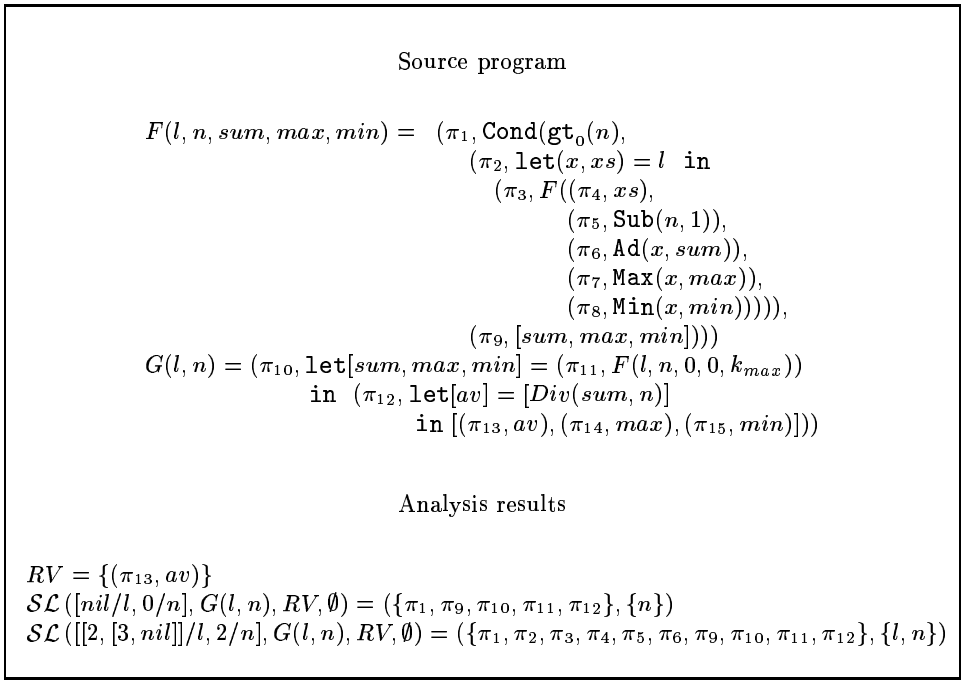

Figure 20: Slicing: dynamic analysis of a simple functional program

7 Related work

Few papers have been devoted to the semantics of program slicing so far. [15] proves a relationship between the behaviour of the original program and the behaviour of the slice. Their definition is close to our $\mathcal{R}_b$ relation[13] but the semantics of the language is expressed in terms of program dependence graphs in [15]; so the programs are first analysed in order to extract their dependences. This approach is well suited to the treatment of imperative languages. It can possibly be extended to other classes of languages, but it might be necessary to generalise the notion of dependence graph as well. In contrast, our framework is not biased towards one specific class of languages. [21] provides formal definitions and a classification of different notions of slicing. The main distinctions are backward vs forward analysers, executable vs non executable slices, and dynamic vs static analysers. Their definitions are based on denotational semantics and they focus on the specifications of the analyses. The goal of the work reported in [4] is probably the closest to ours. The authors describe a family of slicing algorithms generalising the notions of dynamic and static slice to that of a constrained slice. Genericity with respect to the programming language is achieved through a translation into an intermediate representation called PIM. Programs are represented as directed acyclic graphs whose semantics is defined in terms of rewriting rules. Slicing is carried out using term graph rewriting with a technique for tracing dynamic dependence relations. So, the main departure of our work with respect to [4] is the fact that our starting point is the natural semantics of the language, which, we believe, leads to a more direct treatment (avoiding a translation into an intermediate language). It should be noted that a richer notion of slicing has been proposed for logic programming languages, which returns not only the set of program points that must be kept in the slice, but also the necessary variables at each program point [19]. This increased precision can also be expressed in our framework (and is useful for other classes of languages) but we preferred to present the simpler version here for the sake of size and readability.

Generic frameworks for program analysis have been proposed in the context of logic programming languages [7] and data flow analysis [22, 24]. They rely on abstract interpretations of denotational semantics ([7, 22]) or interpreters ([24]) and genericity is achieved by parameterising the abstract domains and choosing appropriate abstract functions. The implementation details of the analysis algorithm can be factorised. While these tools may attain a higher degree of mechanisation than our framework, they do not offer to the user the same level of abstraction: they take as input the *specification of an abstract interpreter* rather than the *specification of a property*. In contrast, we have chosen to specify a program analysis by the transformation that it enables. This choice is motivated by our belief that a specification in terms of program transformation is closer to the ultimate purpose of the analysis. Despite this difference of point of view, all these works are obviously inspired by the same goals.

Most program analyses in the past were based on either low-level operational semantics [2] or denotational semantics [10], but the use of structural operational semantics or natural semantics as a basis for the design of analysers has been advocated by several researchers recently [8, 13, 18, 17]. A notion of Structural Operational Semantics format was introduced in [16] in order to establish a metatheory supporting general reasoning techniques. The broad objective of this paper is similar to ours (defining semantics formats to achieve a level of factorisation over a class of programming languages) but the context is different: the format of [16] is dedicated to functional programming languages and the paper is concerned with the establishment of general proof techniques (like fixed-point induction) rather than program analysis.

[13] One difference comes from the fact that their slicing criterion is a program point rather than a function from program points to sets of variables.

8 Conclusion

The overall goal of our work is to achieve a higher level of factorisation in the design of program analyses. We believe that slicing analysis is a good illustration of the advantages of our approach because its goal is universal (it does not depend on the idiosyncrasies of a particular language (even if the features of the language obviously affect the derived analysis) and it has already been studied in the context of imperative languages and (more recently) logic and functional programming languages. Constructing a correct slicing analyser for a logic programming language from scratch is clearly not a trivial task [19], so the benefit of our approach is striking in this case. As mentioned in section 3, the framework presented here accommodates forward analyses as well as backward analyses. As an example of forward analysis, we have defined a generic sharing analysis (with the associated compile-time garbage collection optimisation) and we have instantiated it to the three major classes of languages (imperative, logic and functional).

We have focussed on dynamic analysis in the body of paper. Our generic dynamic analyser is defined in a strongly typed functional language[14]. As a consequence, we can rely on previous results on logical relations and abstract interpretation [1, 2] in order to systematically construct static analysers from the dynamic analysers. We first must provide abstract domains for the static slicing analyser and the corresponding abstraction functions. We recall that the type of the dynamic analyser is $\mathtt{C} \times \mathtt{T} \times (\mathtt{PP} \rightarrow \mathcal{P}(\mathtt{Pvar})) \times \mathcal{P}(\mathtt{Pvar}) \rightarrow \mathcal{P}(\mathtt{PP}) \times \mathcal{P}(\mathtt{Pvar})$. Since $\mathtt{PP} \rightarrow \mathcal{P}(\mathtt{Pvar})$, $\mathcal{P}(\mathtt{Pvar})$ and $\mathcal{P}(\mathtt{PP})$ are already abstract domains associated with the dynamic analysis, only C needs to be abstracted[15]. The next stage to derive a correct static analyser is to find appropriate abstractions for the constants and operators occurring in the definition of the analyser ($\mathcal{C}_i^k, \mathcal{T}_i^k, \ldots$). It is shown in [1] that the correctness of the abstract interpretation of the constants and operators of the language entails the correctness of the abstract interpretation of the whole language. The correctness of the abstract interpretation means that the results of the dynamic analysis and the static analysis are related if their arguments are. In fact, it is possible to define the most precise abstraction for each constant and operator of the language [1]. The basic idea to find the best abstraction $op^a(v_1^a, \ldots, v_n^a)$ of an operator op is to define it as the least upper bound of the abstractions of all the results of op applied to arguments v_i belonging to the concretisation sets of the arguments of the v_i^a. The technique sketched here provides a systematic way to construct a correct abstract interpretation, and thus to derive a static analyser from a dynamic analyser. By deriving static analysers as abstractions of dynamic analysers, we can see the dynamic analyser either as an intermediate stage in the derivation of a static analyser (playing a role similar to a collecting semantics) or as the final product of the derivation. This technique has been applied in [5] to obtain static slicing analysers for an imperative language and a functional language.

There are two main aspects in which the work described here may seem limited: we have used only natural semantics and first-order languages. We review these issues in turn:

- Structural Operational Semantics [12] are more precise than natural semantics and they are required for a proper treatment of non-determinism, non-termination and parallelism [11]. In fact, the natural semantics format introduced in section 2 can be adapted to Structural Operational Semantics without difficulty[16] and the dynamic analyses can be defined in the very same way for Structural Operational Semantics. The added difficulty introduced by Structural Operational Semantics is the fact that they create new program fragments which makes it necessary to abstract over the syntax of the language to derive a static analysis. This problem is discussed in [17].

[14] Note that the typing mentioned here has nothing to do with the language in which the analysed programs are written, this language itself can perfectly well be untyped.

[15] Of course, as usual in abstract interpretation, $\mathtt{PP} \rightarrow \mathcal{P}(\mathtt{Pvar})$, $\mathcal{P}(\mathtt{Pvar})$ and $\mathcal{P}(\mathtt{PP})$ can also be abstracted if further approximations are needed, but we do not consider this issue here.

[16] In order to deal with Structural Operational Semantics, we basically need to change the type NF in STT and to introduce a global loop in the semantics format since a Structural Operational Semantics rule represents a single evaluation step.

- We contented ourselves with the analysis of first-order languages in this paper, but we have shown in [5] how to deal with (higher-order) analyses of higher-order languages. As far as the semantics format is concerned, we need to extend $\mathcal{C}_i^k$ to include the subterms U_i in its arguments. The technique for constructing the static analysis consists in deriving a function returning a pair of values: the first element of the pair represents the first-order information and the second element is the higher-order component. Because of this decomposition, the treatments are significantly more complex (even if the principles are the same), so we preferred to avoid these extra complications in this paper. Depending on the importance attached to higher-order analyses, this may be considered as a weakness of our approach.

References

[1] S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, 1:5–40, 1990.

[2] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth annual ACM Symposium on Principles of Programming Languages, POPL'77*, pages 238–252, Los Angeles, California, January 1977.

[3] T. Despeyroux. Typol: a formalism to implement natural semantics. Technical Report 94, INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le chesnay Cedex FRANCE, 1988.

[4] J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In *22nd annual ACM Symposium on Principles of Programming Languages, POPL'95*, pages 379–392, San Francisco, California, January 1995.

[5] V. Gouranton. *Derivation of dynamic and static analysers by folding/unfolding of operational specifications.* PhD thesis, Rennes, France, September 1997. `http://www.irisa.fr/EXTERNE/bibli/Centredoc_anglais.html` .

[6] G. Kahn. Natural semantics. In *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science*, number 247 in Lecture Notes in Computer Science, pages 22–39, Passau, Germany, 1987. Springer-Verlag.

[7] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A generic interpretation algorithm and its complexity analysis. In koichi Furukawa, editor, *Proceedings of the 8th International Conference on Logic Programming*, pages 64–78, Paris, France, 1991.

[8] D. Le Métayer and D. Schmidt. Structural operational semantics as a basis for static program analysis. In *ACM Computing Surveys*, volume 28, pages 340–343, 1996.

[9] D. Miller and G. Nadathur. Higher-order logic programming. In E. Shapiro, editor, *Third International Logic Programming Conference*, volume 225 of *Lecture Notes in Computer Science*, pages 448–462, Imperial College, London, United Kingdom, July 1986.

[10] F. Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.

[11] H. Riis Nielson and F. Nielson. *Semantics With Applications.* John Wiley & Sons, 1992.

[12] G.D. Plotkin. A structural approach of operational semantics. Technical Report FN-19, DAIMI, Aarhus University, Denmark, 1981.

[13] S. Purushothaman and J. Seaman. From operational definitions to abstract semantics. In *Conference on Functional programming Languages and Computer Architecture, FPCA'93*, pages 276–285, Copenhagen, Denmark, June 1993.

[14] T. Reps and T. Turnidge. Program specialization via program slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *International Seminar on Partial Evaluation*, Dagstuhl Castle, Germany, February 1996.

[15] T. Reps and W. Yang. The semantics of program slicing. Technical Report 777, University of Wisconsin, Madison, 1988.

[16] D. Sands. From sos rules to proof principles: an operational metatheory for functional languages. In *24th Annual ACM Symposium on Principles of Programming Languages, POPL'97*, Paris, France, January 1997.

[17] D. Schmidt. Abstract interpretation of the operational semantics hierarchy. Technical report, Kansas State University, 1997.

[18] D.A. Schmidt. Natural-semantics-based abstract interpretation. In *Second international Static Analysis Symposium, SAS'95*, number 983 in Lecture Notes in Computer Science, Glasgow, UK, September 1995. Springer-Verlag.

[19] S. Schoenig and M. Ducassé. A backward slicing algorithm for Prolog. In *Third International Static Analysis Symposium, SAS'96*, number 1145 in Lecture Notes in Computer Science, pages 317–331, Aachen, Germany, September 1996. Springer-Verlag.

[20] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[21] G. A. Venkatesh. The semantics approach to program slicing. In *the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, PLDI'91*, pages 107–119, Toronto, Ontario, Canada, June 1991.

[22] G.A. Venkatesh. A framework for construction and evaluation of high-level specifications for program analysis techniques. In *SIGPLAN Conference on Programming Language Design and Implementation, PLDI'89*, volume 24, pages 1–12, Portland, Oregon, July 1989.

[23] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 4:352–357, July 1984.

[24] K. Yi and W. L. Harrison III. Automatic generation and management of interprocedural program analyses. In *Twentieth Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'93*, pages 246–259, Charleston, South Carolina, January 1993.

Appendix 1

Correctness proof of the generic slicing analyser

The correctness property of the generic slicing analyser of Figure 6 was stated as follows in section 3:

DEFINITION **3.1** Correctness of the generic slicing analysis

$$\begin{array}{ll} \text{If} & \mathcal{SL}\,(C,T,RV,D) = (S,N) \\ & C =_N C' \\ & T' = \mathcal{A}nn(S,T) \\ \\ \text{then} & \mathcal{R}_b\,(RV, \mathcal{S}\,(C,T), \mathcal{S}\,(C',T')) \\ & \mathcal{S}\,(C,T).\texttt{stt.nf} =_D \mathcal{S}\,(C',T').\texttt{stt.nf} \end{array}$$

The proof of correctness of the analyser of Figure 6 can be achieved by a double recurrence on the structure of the natural semantics derivation tree. The first recurrence is on the depth of the derivation tree and the second recurrence is on the list of subtrees of a node in the natural semantics derivation tree.

It is easy to see that both orderings are well-founded if the natural semantics derivation tree is finite (in other words, we consider the analysis of terminating programs only).

Properties (1) and (2) were defined as follows in section 3:

$$\text{for all } k \in [1,n] : C_k =_{N_k} C'_k \quad \text{and} \quad T'_k = \mathcal{A}nn(S_k,T_k) \quad \text{implies} \quad \mathcal{R}_b\,(RV, PT_k, PT'_k) \quad \text{and} \quad R_k =_{D_k} R'_k \quad (1)$$

$$\mathcal{R}_b\,(RV, PT, PT') \quad \text{and} \quad \mathcal{F}^k(C,\overline{R},E) =_D \mathcal{F}^k_\theta(C',\overline{R}',E) \quad \text{and} \quad \mathcal{P}^k(C,E) \Rightarrow \mathcal{P}^k_\theta(C',E) \qquad (2)$$

Following the above proof skeleton, we prove property (2) by recurrence on the depth of the derivation tree using the induction hypothesis (1).

We consider successively the two cases in the definition of $\mathcal{SL}$ (Figure 6):

- $\mathcal{SL}(C,T,RV,D) = (\ +_i\ S_i + \{\pi\}, N)$
- $\mathcal{SL}(C,T,RV,D) = (\emptyset, D\ +\ \mathcal{N}'_0\,(P^k))$

1.1 Case $\mathcal{SL}(C,T,RV,D) = (+_i\ S_i + \{\pi\}, N)$

We first show the following property by recurrence:

$$\forall\ k.\ C_k =_{N_k} C'_k \text{ and } T'_k = \mathcal{A}nn(S_k,T_k) \qquad (3)$$

We assume the induction hypothesis:

$$\text{for all } j < i : C_j =_{N_j} C'_j \text{ and } T'_j = \mathcal{A}nn(S_j,T_j) \qquad (4)$$

and we prove:

$$C_i =_{N_i} C'_i \text{ and } T'_i = \mathcal{A}nn(S_i,T_i) \qquad (5)$$

This conjunction is proven in two stages:

1. We first prove $C_i =_{N_i} C'_i$. From Figure 6, we have $C_i = \mathcal{C}^k_i(C,\overline{R},E)$, $C'_i = \mathcal{C}^k_i(C',\overline{R'},E)$, and:

$$N = +_j\ \mathcal{N}_0\,(\mathcal{C}^k_j, N_j)\ \ +_j\ \ \mathcal{N}'_0\,(\mathcal{T}^k_j)\ \ +\ \ \mathcal{N}_0\,(\mathcal{F}^k, D')\ \ +\ \ \mathcal{N}'_0\,(P^k)$$

We have $C =_N C'$ (assumption of Definition 3.1), which implies $C_i =_{N_i} \mathcal{C}_i^k(C', \overline{R}, E)$ since $\mathcal{N}_0(\mathcal{C}_i^k, N_i) \subseteq N$[17]. From (4) and (1), we have $\forall j < i.\ R_j =_{D_j} R'_j$. Since $\mathcal{N}_j(\mathcal{C}_i^k, N_i) \subseteq D_j$ (Figure 6), we get[18]

$$\mathcal{C}_i^k(C', \overline{R}, E) =_{N_i} \mathcal{C}_i^k(C', \overline{R'}, E) = C'_i$$

So, we have shown $C_i =_{N_i} C'_i$.

2. We then prove the second property: $T'_i = \mathcal{A}nn(S_i, T_i)$. The same reasoning applies to this proof: $C =_N C'$ implies $T_i = \mathcal{T}_i^k(C', \overline{R}, T)$ because $\mathcal{N}'_0(\mathcal{T}_i^k) \subseteq N$. Furthermore, $\forall j < i.\ R_j =_{D_j} R'_j$ and $\mathcal{N}_j(\mathcal{T}_i^k) \subseteq D_j$ entail :

$$\mathcal{T}_i^k(C', \overline{R}, T) = \mathcal{T}_i^k(C', \overline{R'}, T)$$

We have $T' = \mathcal{A}nn(+_i\ S_i + \{\pi\}, T)$ (assumption of Definition 3.1). From the compositionality property

$$\mathcal{T}_i^k(C', \overline{R'}, T[U'_m/U_m]) = \mathcal{T}_i^k(C', \overline{R'}, T)[U'_m/U_m]$$

and the definition of $\mathcal{A}nn$, we can derive $\mathcal{T}_i^k(C', \overline{R'}, T') = \mathcal{A}nn(+_i\ S_i + \{\pi\}, \mathcal{T}_i^k(C', \overline{R'}, T))$. We have $\forall i. S_i \leq +_i\ S_i + \{\pi\}$ and using the definition of $\mathcal{A}nn$, we get:

$$\mathcal{T}_i^k(C', \overline{R'}, T') = \mathcal{A}nn(t_i, \mathcal{T}_i^k(C', \overline{R'}, T))$$

Since we have shown $T_i = \mathcal{T}_i^k(C', \overline{R'}, T)$, this allows us to conclude $T'_i = \mathcal{A}nn(S_i, T_i)$.

We have shown both properties of (5), which concludes the proof of (3).

We prove now the three properties of (2) in turn, using (3) and the induction hypothesis (1):

1. We first prove $\mathcal{R}_b(RV, PT, PT')$. From (1) and (4), we have $\forall k \in [1, ...n]\ \mathcal{R}_b(RV, PT_k, PT'_k)$. Since

$$PT = ((C, T, \mathcal{F}^k(C, \overline{R}, E)), \overline{PT}, k) \qquad and$$
$$PT' = ((C', T', \mathcal{F}_\theta^k(C', \overline{R'}, E)), \overline{PT'}, k)$$

we are left with showing $\mathcal{F}^k(C, \overline{R}, E) =_{RV(\pi)} \mathcal{F}_\theta^k(C', \overline{R'}, E)$. We know that :

$$N = +_j\ \mathcal{N}_0(\mathcal{C}_j^k, N_j)\ +_j\ \mathcal{N}'_0(\mathcal{T}_j^k)\ +\ \mathcal{N}_0(\mathcal{F}^k, D + RV(\pi))\ +\ \mathcal{N}'_0(P^k)$$

So $\mathcal{N}_0(\mathcal{F}^k, D + RV(\pi)) \subseteq N$ and, since $C =_N C'$, we have

$$\mathcal{F}^k(C, \overline{R}, E) =_{RV(\pi)} \mathcal{F}^k(C', \overline{R}, E)$$

We also have $\forall j.\ R_j =_{D_j} R'_j$ (from (1) and (4)) and $\mathcal{N}_j(\mathcal{F}^k, D + RV(\pi)) \subseteq D_j$. So, $\mathcal{F}^k(C', \overline{R}, E) =_{RV(\pi)} \mathcal{F}^k(C', \overline{R'}, E)$. We have $(+_i\ S_i + \{\pi\})(\pi) = true$, but $(+_i\ S_i + \{\pi\})(\pi) \leq \theta$, so $\theta = true$ and, since $\mathcal{F}_{true}^k = \mathcal{F}^k$, we can conclude

$$\mathcal{F}^k(C, \overline{R}, E) =_{RV(\pi)} \mathcal{F}^k(C', \overline{R'}, E)$$

So, we have shown $\mathcal{R}_b(RV, PT, PT')$.

[17] See the characterisation of $\mathcal{N}_i$ at the bottom of Figure 6.

[18] Note that it is enough to have $R_j =_{D_j} R'_j$ for $j < i$ because, due to the convention explained in section 2, $\mathcal{C}_i^k$ does not use its arguments $R_i, \ldots, R_n$.

2. We then prove $\mathcal{F}^k(C,\overline{R},E) =_D \mathcal{F}^k_\theta(C',\overline{R}',E)$. From $C =_N C'$ and $\mathcal{N}_0(\mathcal{F}^k, D + RV(\pi)) \subseteq N$, we have
$$\mathcal{F}^k(C,\overline{R},E) =_D \mathcal{F}^k(C',\overline{R},E)$$
From $\forall j.\ R_j =_{D_j} R'_j$ and $\mathcal{N}_j(\mathcal{F}^k, D + RV(\pi)) \subseteq D_j$, we derive
$$\mathcal{F}^k(C',\overline{R},E) =_D \mathcal{F}^k(C',\overline{R'},E)$$
This implies $\mathcal{F}^k(C,\overline{R},E) =_D \mathcal{F}^k(C',\overline{R'},E)$ since $\mathcal{F}^k_{true} = \mathcal{F}^k$.

3. Finally, we prove $\mathcal{P}^k(C,E) \Rightarrow \mathcal{P}^k_\theta(C',E)$. From $C =_N C'$ and $\mathcal{N}'_0(P^k) \subseteq N$, we have
$$P^k(C,E) = P^k(C'E) = P^k_\theta(C',E)$$
because $P^k_{true} = P^k$.

The three properties of (2) are proven. This concludes the first case of the definition of the generic slicing $\mathcal{SL}$. We prove now (2) for the second case.

1.2 Case $\mathcal{SL}(C,T,RV,D) = (\emptyset, D\ +\ \mathcal{N}'_0(P^k))$

We consider the three properties of (2) in turn.

1. The first property $\mathcal{R}_b(RV,PT,PT')$ is satisfied because
$$\forall n \in Nodes(PT).\ RV(PT[n].\texttt{t.pp}) = \emptyset$$
This property can be checked by a simple recurrence on the derivation trees PT_i (the property holds for the root node and the condition in the test of Figure 6 entails $\forall i.S_i = \emptyset$, since $+_i\ S_i = \emptyset$).

2. In order to establish the second property, we first show $\mathcal{F}^k(C,\overline{R},E) =_D C$. We prove this equivalence by a double recurrence on the derivation tree as we did in section 1.1. The test in $\mathcal{SL}$ implies $\forall i.\ S_i = \emptyset$, so we can use the induction hypothesis $\forall i.\ R_i =_{D_i} C_i$. From $\mathcal{A}(\mathcal{F}^k, D)$ and the definition of $\mathcal{F}^k$, we know that $\mathcal{F}^k(C,\overline{R},E) =_D C$ or $\mathcal{F}^k(C,\overline{R},E) =_D R_{i_1}$.

   In the first case, we immediately have the conclusion.

   In the second case, we can infer $\mathcal{N}_{i_1}(\mathcal{F}^k, D) = D$ from the definition of $\mathcal{N}_{i_1}$, so we have $D \subseteq D_{i_1}$. Now, we have $R_{i_1} =_{D_{i_1}} C_{i_1}$ from the induction hypothesis and $D_{i_1} \subseteq N_{i_1}$ from the definition of $\mathcal{SL}$. Since the test in the definition of $\mathcal{SL}$ implies $\mathcal{A}(\mathcal{C}^k_{i_1}, N_{i_1})$, we can infer that $\mathcal{C}^k_{i_1}(C,\overline{R},E) =_{N_{i_1}} C$ or $\mathcal{C}^k_{i_1}(C,\overline{R},E) =_{N_{i_1}} R_{i_2}$ with $i_2 < i_1$[19].

   In the first case, we have $\mathcal{F}^k(C,\overline{R},E) =_D R_{i_1}$ and $R_{i_1} =_{D_{i_1}} C_{i_1} =_{N_{i_1}} C$, which allows us to conclude $\mathcal{F}^k(C,\overline{R},E) =_D C$ since $D \subseteq D_{i_1} \subseteq N_{i_1}$.

   In the second case, we use the (inner) induction hypothesis $R_{i_2} =_D C$, which leads to the same conclusion. So we have established $\mathcal{F}^k(C,\overline{R},E) =_D C$.

   In order to prove
$$\mathcal{F}^k(C,\overline{R},E) =_D \mathcal{F}^k_\theta(C',\overline{R}',E),$$
   we observe that $\emptyset(\pi) \leq \theta$ (from (2)) and $\emptyset(\pi) = \mathit{false}$ here. So we have to examine the two possible cases $\theta = \mathit{false}$ and $\theta = \mathit{true}$.

   If $\theta = \mathit{false}$, then $\mathcal{F}^k_\theta(C',\overline{R}',E) = C' =_D C$ (from (2)) and $\mathcal{F}^k(C,\overline{R},E) =_D C$; so we get the required equivalence.

   If $\theta = \mathit{true}$, then $\mathcal{F}^k_\theta(C',\overline{R}',E) = \mathcal{F}^k(C',\overline{R}',E) =_D C' =_D C = \mathcal{F}^k(C,\overline{R},E)$, which leads again to the required equivalence. This concludes the proof of the second property of (2).

[19] The property $i_2 < i_1$ is guaranteed by the convention taken in section 2 about the natural semantics format.

3. In order to prove the third property of (2), we distinguish the two cases $\theta = \mathit{false}$ and $\theta = \mathit{true}$ as above.

   In the first case, we immediately have $\mathcal{P}^k(C,E) \Rightarrow \mathcal{P}^k_\theta(C',E)$ since $\mathcal{P}^k_{\mathit{false}}(C',E) = \mathit{true}$.

   For the second case, we observe that $C =_N C'$ and $\mathcal{N}'_0(\mathcal{P}^k) \subseteq N$; so

$$\mathcal{P}^k(C,E) = \mathcal{P}^k(C',E) = \mathcal{P}^k_{\mathit{true}}(C',E)$$

We have shown the three properties of (2) for the second case of the definition of $\mathcal{SL}$, which concludes the proof of the correctness of the generic analysis.

Appendix 2

Slicing analysis for a logic programming language

We have presented in the body of the paper (section 5) the instantiation of the generic dynamic slicing analysis for a logic programming language. We show here the different tables used for this instantiation. We first present in Figure 21 the instantiation of the natural semantics format from the Figure 1.

	$\mathcal{C}_1^k$	$\mathcal{C}_2^k$	$\mathcal{T}_1^k$	$\mathcal{T}_2^k$	$\mathcal{F}^k$	P^k
k_1					$\overline{op}(C, x_1, x_2, x_3)$	**true**
k_2					$\overline{unif}\ (C, x, t)$	**true**
k_3	C	R_1	U_1	U_2	R_2	**true**
k_4	C	C	U_1	U_2	$union\,(C, R_1, R_2)$	**true**
k_5	$\overline{Add}\ (C, x, rx)$		U_1		$\overline{Drop}\ (R_1, x)$	**true**
k_6	$\overline{Ren_k}(C)$		B_k		$\overline{Ext_k}(C, R_1)$	**true**

Figure 21: Instantiation of the natural semantics for the logic programming language

To obtain the dynamic slicing analyser, we must define the functions $\mathcal{N}_j$, $\mathcal{N}'_j$ and the predicate $\mathcal{A}$. Their definitions are presented in Figure 22 and Figure 23.

Figure 24 gathers the results that were used to obtain the instantiation for the slicing analyser of Figure 14.

	$\mathcal{N}_0\,(\mathcal{C}_1^k, N_1)$	$\mathcal{N}_0\,(\mathcal{C}_2^k, N_2)$	$\mathcal{N}'_0\,(\mathcal{T}_1^k)$	$\mathcal{N}'_0\,(\mathcal{T}_2^k)$	$\mathcal{N}_0\,(\mathcal{F}^k, D')$	$\mathcal{N}'_0\,(P^k)$
k_1					$D' + \{x_1, x_2, x_3\}$	$\emptyset$
k_2					$D' + Pv(t) + \{x\}$	$\emptyset$
k_3	N_1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
k_4	N_1	N_2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
k_5	$N_1 - \{x\}$		$\emptyset$		$\emptyset$	$\emptyset$
k_6	$\{y_i \mid x_i \in N_1\}$		$\emptyset$		$\{y_1, \ldots, y_n\} + D'$	$\emptyset$

	$\mathcal{N}_1\,(\mathcal{C}_1^k, N_1)$	$\mathcal{N}_1\,(\mathcal{C}_2^k, N_2)$	$\mathcal{N}'_1\,(\mathcal{T}_1^k)$	$\mathcal{N}'_1\,(\mathcal{T}_2^k)$	$\mathcal{N}_1\,(\mathcal{F}^k, D')$
k_3	$\emptyset$	N_2	$\emptyset$	$\emptyset$	$\emptyset$
k_4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	D'
k_5	$\emptyset$		$\emptyset$		D'
k_6	$\emptyset$		$\emptyset$		$\{x_i \mid \neg Indep(C, \{y_i\}, D')\}$

	$\mathcal{N}_2\,(\mathcal{C}_1^k, N_1)$	$\mathcal{N}_2\,(\mathcal{C}_2^k, N_2)$	$\mathcal{N}'_2\,(\mathcal{T}_1^k)$	$\mathcal{N}'_2\,(\mathcal{T}_2^k)$	$\mathcal{N}_2\,(\mathcal{F}^k, D')$
k_3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	D'
k_4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	D'

Figure 22: Instantiation of $\mathcal{N}_i$ for the logic programming language

We provide a justification for some of the values of Figure 22 and Figure 23.

- $\mathcal{N}_0\,(\mathcal{C}_1^{k_6}, N_1) = \{y_i \mid x_i \in N_1\}$

  We have, $\mathcal{C}_1^{k_6}(C, \overline{R}, E) = \overline{Ren_k}(C)$. Following the definition of $\mathcal{N}_i$ (bottom of Figure 6), we have:

$$\begin{array}{l} \mathcal{N}_0\,(\mathcal{C}_1^{k_6}, N_1) = N_0 \Rightarrow \\ (C' =_{N_0} C \Rightarrow \overline{Ren_k}(C') =_{N_1} \overline{Ren_k}(C)) \Rightarrow \\ \forall z \in N_0,\ C'\ z = C\ z \Rightarrow \forall y \in N_1, \overline{Ren_k}(C')\ y = \overline{Ren_k}(C)\ y \\ \qquad\qquad \Rightarrow \forall y \in N_1\ [\theta'_i(y_i)/x_i]\ y = [\theta_i(y_i)/x_i]\ y \end{array}$$

If $y \neq x_i$, then y is not in the considered substitution domain and $y = y$.
If $y = x_i$, then we must verify $\theta'(y_i) = \theta(y_i)$ and this equation is satisfied if $y_i \in N_0$.
It is sufficient to have $N_0 = \{y_i \mid x_i \in N_1\}$.

- $\mathcal{N}_0\,(\mathcal{F}^{k_2}, D') = D' + Pv(term) + \{x\}$
  We have, $\mathcal{F}^{k_2}(C, \overline{R}, E) = \overline{unif}\,(C, x, term)$, and following the definition of $\mathcal{N}$:

$$\begin{array}{l}\mathcal{N}_0\,(\mathcal{F}^{k_2}, D') = N_0 \Rightarrow \\ (C' =_{N_0} C \Rightarrow \overline{unif}\,(C', x, term) =_{D'} \overline{unif}\,(C, x, term)) \Rightarrow \\ \forall z \in N_0,\ C'\ z = C\ z \Rightarrow \forall y \in D', \overline{unif}\,(C', x, term)\ y = \overline{unif}\,(C, x, term)\ y\end{array}$$

  The proof of this property is by induction over the structure of C (definition of the function $\overline{F}$). The induction case is straightforward. For the base case, we have to prove: $\forall z \in N_0,\ \theta'\ z = \theta\ z \Rightarrow \forall y \in D', \Delta(\theta', x, term)\ y = \Delta(\theta, x, term)\ y$
  We must prove that both the conditions and the results are the same. First, the conditions are equivalent, that is to say: ($\theta'(x)$ and $\theta'(term)$ can be unified) $\Leftrightarrow$ ($\theta(x)$ and $\theta(term)$ can be unified). If $x \in N_0$ and $Pv(term) \in N_0$, then $\theta'(x) = \theta(x)$, $\theta'(term) = \theta(term)$ and the conditions are equal. The results are also equal:

$$\begin{array}{rl}\forall y \in D', & \texttt{let}\ \sigma' = mgu(\theta'(x), \theta'(term)) \in \texttt{Rsubst} \\ & \texttt{in}\ (\sigma' \circ \theta')(y) \\ = & \\ & \texttt{let}\ \sigma = mgu(\theta(x), \theta(term)) \in \texttt{Rsubst} \\ & \texttt{in}\ (\sigma \circ \theta)(y)\end{array}$$

  If we choose $Pv(term) + \{x\} \subseteq N_0$ and $\sigma' = \sigma$, it is sufficient to check: $\forall y \in D',\ \theta'(y) = \theta(y)$. So, we have $N_0 = D' + Pv(term) + \{x\}$.

- $\mathcal{N}_1\,(\mathcal{F}^{k_6}, D') = \{x_i \mid \neg Indep(C, \{y_i\}, D')\}$
  We have, $\mathcal{F}^{k_6}(C, \overline{R}, E) = \overline{Ext_k}(C, R_1)$:

$$\begin{array}{l}\mathcal{N}_1\,(\mathcal{F}^{k_6}, D') = N_1 \Rightarrow \\ (R'_1 =_{N_1} R''_1 \Rightarrow \overline{Ext_k}(C, R'_1) =_{D'} \overline{Ext_k}(C, R''_1))\end{array}$$

  For the base case, we have:

$$\begin{array}{l}\forall z \in N_1,\ \theta'\ z = \theta''\ z \Rightarrow \\ \forall y \in D', Ext_k(\theta, \theta')(y) = Ext_k(\theta, \theta'')(y)\end{array}$$

$$((\sigma' \circ \theta)\ y \ \text{ with } \theta' = \sigma' \circ Ren_k(\theta)) =_{D'} ((\sigma'' \circ \theta)\ y \ \text{ with } \theta'' = \sigma'' \circ Ren_k(\theta))$$

  It is sufficient to have: $\forall rv \in Dom(\theta_{/D'}),\ \sigma'(rv) = \sigma''(rv)$, which holds if $\theta' =_{N_1} \theta''$ with $N_1 = \{x_i \mid \neg Indep(\theta, \{y_i\}, D')\}$. This is because if $\exists rv \in Dom(\theta_{/D'})$ with $\sigma'(rv) \neq \sigma''(rv)$ then $\exists y_i$ with $rv \in \theta(y_i)$ and $\exists x_i$ with $\neg Indep(\theta, \{y_i\}, D')$ and $\theta'(x_i) \neq \theta''(x_i)$. From then, we get $\theta' \neq_{N_1} \theta''$, which is a contradiction. We choose: $N_1 = \{x_i \mid \neg Indep(C, \{y_i\}, D')\}$, because the induction case is straightforward.

- Following the definition of $\mathcal{A}$ and for k_1, we have: $\mathcal{A}\,(\mathcal{F}^{k_1}, D) \Rightarrow \mathcal{F}^{k_1}(C, \overline{R}, E) =_D C$
  that is to say: $\mathcal{A}\,(\mathcal{F}^{k_1}, D) \Rightarrow \forall z \in D\ \overline{\texttt{B(Op)}}\,(C, x_1, x_2, x_3)\ z = C\ z$. It is sufficient to check this property for $C = \theta$:

$$\begin{array}{rl}\mathcal{A}\,(\mathcal{F}^{k_1}, D) & \Rightarrow \forall z \in D\ \texttt{B(Op)}\,(\sigma, x_1, x_2, x_3)\ z = \theta\ z \\ & \Rightarrow \forall z \in D\ \ (\texttt{let}\ \sigma = [(\theta(x_1)\ Op\ \theta(x_2))/\theta(x_3)] \\ & \qquad\qquad \texttt{in}\ \sigma \circ \theta(z) \\ & \qquad\qquad = \theta\ (z))\end{array}$$

	$\mathcal{A}(\mathcal{F}^k, D)$	$\mathcal{A}(\mathcal{C}_1^k, N_1)$	$\mathcal{A}(\mathcal{C}_2^k, N_2)$
k_1	$Indep(C, D, \{x_3\})$		
k_2	$UF(C, x, t)$ and $Indep(C, D, Pv(t) + \{x\})$		
k_3	**true**	**true**	**true**
k_4	**false**	**true**	**true**
k_5	**true**	**true**	
k_6	$Indep(C, D, \{y_1, \ldots, y_n\})$	$N_1 = \emptyset$	

Figure 23: Instantiation of $\mathcal{A}$ for the logic programming language

	D_1	D_2	N
k_1			$D' + \{x_1, x_2, x_3\}$
k_2			$D' + Pv(t) + \{x\}$
k_3	N_2	D'	N_1
k_4	D'	D'	$N_1 + N_2$
k_5	D'		$N_1 - \{x\}$
k_6	$\{x_i \mid \neg Indep(C, \{y_i\}, D')\}$		$D' + \{y_1, \ldots, y_n\}$

	$\mathcal{A}(\mathcal{F}^k, D)$ and$_{i,\ i=1,2}$ $\mathcal{A}(\mathcal{C}_i^k, N_i)$
k_1	$Indep(C, D, \{x_3\})$
k_2	$UF(C, x, t)$ and $Indep(C, D, Pv(t) + \{x\})$
k_3	**true**
k_4	**false**
k_5	**true**
k_6	$Indep(C, D, \{y_1, \ldots, y_n\})$ and $N_1 = \emptyset$

Figure 24: Final instantiation for the logic programming language

$$\mathcal{A}(\mathcal{F}^{k_1}, D) \Rightarrow \quad (1) \quad \begin{array}{l} \texttt{let } \sigma \ = \ [(\theta(x_1)\ Op\ \theta(x_2))/\theta(x_3)] \\ \texttt{in } (\sigma \circ \theta)(z) \\ = \theta(z) \end{array}$$

Setting $\mathcal{A}(\mathcal{F}^{k_1}, D) = Indep(\theta, D, \{x_3\})$, then (1) is satisfied because the substitution resulting from the evaluation of the term cannot have any impact on the variables of D. Since we have the restriction $C = \theta$, so we set in all cases: $\mathcal{A}(\mathcal{F}^{k_1}, D) = Indep(C, D, \{x_3\})$.

Appendix 3

Slicing analysis for a functional language

Section 6 defined the dynamic slicing for a functional language. We present in this appendix the different tables used to achieve this goal. The Figure 25 presents the instantiation of the natural semantics from Figure 1.

k	$\mathcal{C}_1^k$	$\mathcal{C}_2^k$	$\mathcal{C}_3^k$	$\mathcal{T}_1^k$	$\mathcal{T}_2^k$	$\mathcal{T}_3^k$	$\mathcal{F}^k$	P^k
k_1							$C[b/res]$	**true**
k_2							$C[C(x)/res]$	**true**
k_3	C	C		U_1	U_2		$C[v_{op}/res]$	**true**
k_4	C	C		U_1	U_2		$C[v_{tuple}/res]$	**true**
k_5	C			U_1			R_1	$op(x) =$ **true**
k_6	C			U_2			R_1	$op(x) =$ **false**
k_7	C	C	$[R_1(res)/x,$ $R_2(res)/y]$	U_1	U_2	E	$C[R_3(res)/res]$	**true**
k_8	C	$C[v_1/x_1,$ $v_2/x_2]$		U_1	U_2		$C[R_2(res)/res]$	**true**

$v_{op} = Val(op)(R_1(res), R_2(res))$ (in k_3)
$v_{tuple} = [R_1(res), R_2(res)]$ (in k_4)
$[v_1, v_2] = R_1(res)$ (in k_8)

Figure 25: Instantiation of the natural semantics for the functional programming language

The definitions of the function $\mathcal{N}_j$ and $\mathcal{N}'_j$ are presented in Figure 26 and the definition of the predicate $\mathcal{A}$ is given in Figure 27. We just provide a justification for some of the values of Figure 26.

- $\mathcal{N}_0\,(\mathcal{C}_2^k, N_2) = N_2 - \{x_1, x_2\}$

  We have, from the definition of $\mathcal{N}_i$:

  $\mathcal{N}_0\,(\mathcal{C}_2^k, N_2) = N_i \Rightarrow$
  $(C' =_{N_i} C \Rightarrow \mathcal{C}_2^k(C', \overline{R}, E) =_{N_2} \mathcal{C}_2^k(C, \overline{R}, E)) \Rightarrow$
  $(\forall y \in N_i.\ C'(y) = C(y) \Rightarrow \forall z \in N_2\ C'[v_1/x_1, v_2/x_2](z) = C[v_1/x_1, v_2/x_2](z)$
  with $[v_1, v_2] = R_1(res))$

  It is sufficient to set $N_i = N_2 - \{x_1, x_2\}$ because we have for $z = x_1$ or x_2 :

  $$C'[v_1/x_1, v_2/x_2](x_1) = C' \text{ and } C[v_1/x_1, v_2/x_2](x_1) = C$$
  $$C'[v_1/x_1, v_2/x_2](x_2) = C' \text{ and } C[v_1/x_1, v_2/x_2](x_2) = C$$

  The other values also satisfy the equivalence.

- $\mathcal{N}_0\,(\mathcal{F}^k, D') =$ `if` $res.1 \notin D'$ `then` $\emptyset$ `else` $\{res\}$

  We have:

  $\mathcal{N}_0\,(\mathcal{F}^k, D') = N_i \Rightarrow$
  $(R'_1 =_{N_i} R_1 \Rightarrow \mathcal{F}^k(C, \overline{R'}, E) =_{D'} \mathcal{F}^k(C, \overline{R}, E)) \Rightarrow$
  $(R'_1 =_{N_i} R_1 \Rightarrow C[v'/res] =_{D'} C[v/res]$
  with $v' = (R'_1(res), R_2(res))$ and $v = (R_1(res), R_2(res))$

	$\mathcal{N}_0(\mathcal{C}_1^k, N_1)$	$\mathcal{N}_0(\mathcal{C}_2^k, N_2)$	$\mathcal{N}_0(\mathcal{C}_3^k, N_3)$	$\mathcal{N}'_0(\mathcal{T}_i^k)$	$\mathcal{N}_0(\mathcal{F}^k, D')$	$\mathcal{N}'_0(P^k)$
k_1					D'/res	$\emptyset$
k_2					$D'/res + \{x\}$	$\emptyset$
k_3	N_1	N_2		$\emptyset$	D'/res	$\emptyset$
k_4	N_1	N_2		$\emptyset$	D'/res	$\emptyset$
k_5	N_1			$\emptyset$	$\emptyset$	$\{x\}$
k_6	N_1			$\emptyset$	$\emptyset$	$\{x\}$
k_7	N_1	N_2	$\emptyset$	$\emptyset$	D'/res	$\emptyset$
k_8	N_1	$N_2 - \{x_1, x_2\}$		$\emptyset$	D'/res	$\emptyset$

	$\mathcal{N}_1(\mathcal{C}_1^k, N_1)$	$\mathcal{N}_1(\mathcal{C}_2^k, N_2)$	$\mathcal{N}_1(\mathcal{C}_3^k, N_3)$	$\mathcal{N}'_1(\mathcal{T}_i^k)$	$\mathcal{N}_1(\mathcal{F}^k, D')$
k_3	$\emptyset$	$\emptyset$		$\emptyset$	$\{res\} \cap D'$
k_4	$\emptyset$	$\emptyset$		$\emptyset$	`if` $res.1 \notin D'$ `then` $\emptyset$ `else` $\{res\}$
k_5	$\emptyset$			$\emptyset$	D'
k_6	$\emptyset$			$\emptyset$	D'
k_7	$\emptyset$	$\emptyset$	`if` $x \notin N_3$ `then` $\emptyset$ `else` $\{res\}$	$\emptyset$	$\emptyset$
k_8	$\emptyset$	(`if` $x_1 \notin N_2$ `then` $\emptyset$ `else` $\{res.1\}$)+ (`if` $x_2 \notin N_2$ `then` $\emptyset$ `else` $\{res.2\}$)		$\emptyset$	$\emptyset$

	$\mathcal{N}_2(\mathcal{C}_1^k, N_1)$	$\mathcal{N}_2(\mathcal{C}_2^k, N_2)$	$\mathcal{N}_2(\mathcal{C}_3^k, N_3)$	$\mathcal{N}'_2(\mathcal{T}_i^k)$	$\mathcal{N}_2(\mathcal{F}^k, D')$
k_3	$\emptyset$	$\emptyset$		$\emptyset$	$\{res\} \cap D'$
k_4	$\emptyset$	$\emptyset$		$\emptyset$	`if` $res.2 \notin D'$ `then` $\emptyset$ `else` $\{res\}$
k_7	$\emptyset$	$\emptyset$	`if` $y \notin N_3$ `then` $\emptyset$ `else` $\{res\}$	$\emptyset$	$\emptyset$
k_8	$\emptyset$	$\emptyset$		$\emptyset$	$D'!res$

	$\mathcal{N}_3(\mathcal{C}_1^k, N_1)$	$\mathcal{N}_3(\mathcal{C}_2^k, N_2)$	$\mathcal{N}_3(\mathcal{C}_3^k, N_3)$	$\mathcal{N}_3(\mathcal{T}_i^k)$	$\mathcal{N}_3(\mathcal{F}^k, D')$
k_7	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$D'!res$

Figure 26: Instantiation of $\mathcal{N}_i$ for the functional programming language

	$\mathcal{A}(\mathcal{C}_1^k, N_1)$	$\mathcal{A}(\mathcal{C}_2^k, N_2)$	$\mathcal{A}(\mathcal{C}_3^k, N_3)$	$\mathcal{A}(\mathcal{F}^k, D)$
k_1				$D?res$
k_2				$D?res$
k_3	**true**	**true**		$D?res$
k_4	**true**	**true**		$D?res$
k_5	**true**			**true**
k_5	**true**			**true**
k_7	**true**	**true**	$\{x, y\} \cap N_3 = \emptyset$	$D?res$
k_8	**true**	$\{x_1, x_2\} \cap N_2 = \emptyset$		$D?res$

Figure 27: Instantiation of $\mathcal{A}$ for the functional programming language

	D_1	D_2	D_3
k_3	$res \cap D'$	$res \cap D'$	
k_4	if $res.1 \notin D'$ then $\emptyset$ else $\{res\}$	if $res.2 \notin D'$ then $\emptyset$ else $\{res\}$	
k_5	D'		
k_6	D'		
k_7	if $x \notin N_3$ then $\emptyset$ else $\{res\}$	if $y \notin N_3$ then $\emptyset$ else $\{res\}$	$D'!res$
k_8	(if $x_1 \notin N_2$ then $\emptyset$ else $\{res.1\}$)+ (if $x_2 \notin N_2$ then $\emptyset$ else $\{res.2\}$)	$D'!res$	

	NN	$\mathcal{A}(\mathcal{F}^k, D)$ and$_{i\ (i=1,2,3)}$ $\mathcal{A}(\mathcal{C}_i^k, N_i)$
k_1	D'/res	$D?res$
k_2	$D'/res + \{x\}$	$res \notin D$
k_3	$N_1 + N_2 + D'/res$	$res \notin D$
k_4	$N_1 + N_2 + D'/res$	$D?res$
k_5	$N_1 + \{x\}$	**true**
k_6	$N_1 + \{x\}$	**true**
k_7	$N_1 + N_2 + D'/res$	$D?res$ and $\{x, y\} \cap N_3 = \emptyset$
k_8	$N_1 + (N_2 - \{x_1, x_2\}) + D'/res$	$D?res$ and $\{x_1, x_2\} \cap N_2 = \emptyset$

Figure 28: Final instantiation for the functional programming language

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399